

---

# Corda Developer Documentation

*Release Master*

R3

Sep 25, 2019

## CONTENTS

<b>1</b>	<b>Release notes</b>	<b>2</b>
<b>2</b>	<b>Upgrading apps to Corda 4</b>	<b>13</b>
<b>3</b>	<b>Upgrading your node to Corda 4</b>	<b>25</b>
<b>4</b>	<b>Corda API</b>	<b>27</b>
<b>5</b>	<b>Cheat sheet</b>	<b>128</b>
<b>6</b>	<b>Getting started developing CorDapps</b>	<b>130</b>
<b>7</b>	<b>Key concepts</b>	<b>143</b>
<b>8</b>	<b>CorDapps</b>	<b>180</b>
<b>9</b>	<b>Tutorials</b>	<b>262</b>
<b>10</b>	<b>Tools</b>	<b>370</b>
<b>11</b>	<b>Node internals</b>	<b>395</b>
<b>12</b>	<b>Component library</b>	<b>402</b>
<b>13</b>	<b>Serialization</b>	<b>409</b>
<b>14</b>	<b>JSON</b>	<b>436</b>
<b>15</b>	<b>Troubleshooting</b>	<b>437</b>
<b>16</b>	<b>Nodes</b>	<b>438</b>
<b>17</b>	<b>Networks</b>	<b>493</b>
<b>18</b>	<b>Official Corda Docker Image</b>	<b>546</b>
<b>19</b>	<b>Azure Marketplace</b>	<b>550</b>
<b>20</b>	<b>AWS Marketplace</b>	<b>558</b>
<b>21</b>	<b>Load testing</b>	<b>560</b>
<b>22</b>	<b>Shell extensions for CLI Applications</b>	<b>564</b>

<b>23 Deterministic Corda Modules</b>	<b>566</b>
<b>24 Changelog</b>	<b>572</b>

Corda is an open-source blockchain platform. If you'd like a quick introduction to blockchains and how Corda is different, then watch this short video:

Want to see Corda running? Download our demonstration application [DemoBench](#) or follow our [quickstart guide](#).

If you want to start coding on Corda, then familiarise yourself with the [key concepts](#), then read our [Hello, World! tutorial](#). For the background behind Corda, read the non-technical [platform white paper](#) or for more detail, the technical [white paper](#).

If you have questions or comments, then get in touch on [Slack](#) or ask a question on [Stack Overflow](#).

We look forward to seeing what you can do with Corda!

---

**Note:** You can read this site offline. Either [download the PDF](#) or download the Corda source code, run `gradle buildDocs` and you will have a copy of this site in the `docs/build/html` directory.

---

---

CHAPTER  
ONE

---

## RELEASE NOTES

### Contents

- *Release notes*
  - *Corda 4.1*
  - *Corda 4*

## 1.1 Corda 4.1

It's been a little under 3 1/2 months since the release of Corda 4.0 and all of the brand new features that added to the powerful suite of tools Corda offers. Now, following the release of Corda Enterprise 4.0, we are proud to release Corda 4.1, bringing over 150 fixes and documentation updates to bring additional stability and quality of life improvements to those developing on the Corda platform.

Information on Corda Enterprise 4.0 can be found [here](#) and [here](#). (It's worth noting that normally this document would have started with a comment about whether or not you'd been recently domiciled under some solidified mineral material regarding the release of Corda Enterprise 4.0. Alas, we made that joke when we shipped the first release of Corda after Enterprise 3.0 shipped, so the thunder has been stolen and repeating ourselves would be terribly gauche.)

Corda 4.1 brings the lessons and bug fixes discovered during the process of building and shipping Enterprise 4.0 back to the open source community. As mentioned above there are over 150 fixes and tweaks here. With this release the core feature sets of both entities are far closer aligned than past major releases of the Corda that should make testing your CorDapps in mixed type environments much easier.

As such, we recommend you upgrade from Corda 4.0 to Corda 4.1 as soon possible.

### 1.1.1 Issues Fixed

- Docker images do not support passing a prepared config with initial registration [CORDA-2888]
- Different hashes for container Corda and normal Corda jars [CORDA-2884]
- Auto attachment of dependencies fails to find class [CORDA-2863]
- Artemis session can't be used in more than one thread [CORDA-2861]
- Property type checking is overly strict [CORDA-2860]
- Serialisation bug (or not) when trying to run SWIFT Corda Settler tests [CORDA-2848]
- Custom serialisers not found when running mock network tests [CORDA-2847]

- Base directory error message where directory does not exist is slightly misleading [CORDA-2834]
- Progress tracker not reloadable in checkpoints written in Java [CORDA-2825]
- Missing quasar error points to non-existent page [CORDA-2821]
- TransactionBuilder can build unverifiable transactions in V5 if more than one CorDapp loaded [CORDA-2817]
- The node hangs when there is a dis-connection of Oracle database [CORDA-2813]
- Docs: fix the latex warnings in the build [CORDA-2809]
- Docs: build the docs page needs updating [CORDA-2808]
- Don't retry database transaction in abstract node start [CORDA-2807]
- Upgrade Corda Core to use Java Persistence API 2.2 [CORDA-2804]
- Network map stopped updating on Testnet staging notary [CORDA-2803]
- Improve test reliability by eliminating fixed-duration Thread.sleeps [CORDA-2802]
- Not handled exception when certificates directory is missing [CORDA-2786]
- Unable to run FinalityFlow if the initiating app has targetPlatformVersion=4 and the recipient is using the old version [CORDA-2784]
- Performing a registration with an incorrect Config gives error without appropriate info [CORDA-2783]
- Regression: java.lang.Comparable is not on the default whitelist but never has been [CORDA-2782]
- Docs: replace version string with things that get substituted [CORDA-2781]
- Inconsistent docs between internal and external website [CORDA-2779]
- Change the doc substitution so that it works in code blocks as well as in other places [CORDA-2777]
- net.corda.core.internal.LazyStickyPool#toIndex can create a negative index [CORDA-2772]
- NetworkMapUpdater#fileWatcherSubscription is never assigned and hence the subscription is never cleaned up [CORDA-2770]
- Infinite recursive call in NetworkParameters.copy [CORDA-2769]
- Unexpected exception de-serializing throwable for OverlappingAttachmentsException [CORDA-2765]
- Always log config to log file [CORDA-2763]
- ReceiveTransactionFlow states to record flag gets quietly ignored if checkSufficientSignatures = false [CORDA-2762]
- Fix Driver's PortAllocation class, and then use it for Node's integration tests. [CORDA-2759]
- State machine logs an error prior to deciding to escalate to an error [CORDA-2757]
- Migrate DJVM into a separate module [CORDA-2750]
- Error in HikariPool in the performance cluster [CORDA-2748]
- Package DJVM CLI for standalone distribution [CORDA-2747]
- Unable to insert state into vault if notary not on network map [CORDA-2745]
- Create sample code and integration tests to showcase rpc operations that support reconnection [CORDA-2743]
- RPC v4 client unable to subscribe to progress tracker events from Corda 3.3 node [CORDA-2742]

- Doc Fix: Rpc client connection management section not fully working in Corda 4 [CORDA-2741]
- AnsiProgressRenderrer may start reporting incorrect progress if tree contains identical steps [CORDA-2738]
- The FlowProgressHandle does not always return expected results [CORDA-2737]
- Doc fix: integration testing tutorial could do with some gradle instructions [CORDA-2729]
- Release upgrade to Corda 4 notes: include upgrading quasar.jar explicitly in the Corda Kotlin template [CORDA-2728]
- DJVM CLI log file is always empty [CORDA-2725]
- DJVM documentation incorrect around *djvm check* [CORDA-2721]
- Doc fix: reflect the CorDapp template doc changes re quasar/test running the official docs [CORDA-2715]
- Upgrade to Corda 4 test docs only have Kotlin examples [CORDA-2710]
- Log message “Cannot find flow corresponding to session” should not be a warning [CORDA-2706]
- Flow failing due to “Flow sessions were not provided” for its own identity [CORDA-2705]
- RPC user security using Shiro docs have errant commas in example config [CORDA-2703]
- The crlCheckSoftFail option is not respected, allowing transactions even if strict checking is enabled [CORDA-2701]
- Vault paging fails if setting max page size to *Int.MAX\_VALUE* [CORDA-2698]
- Upgrade to Corda Gradle Plugins 4.0.41 [CORDA-2697]
- Corda complaining of duplicate classes upon start-up when it doesn't need to [CORDA-2696]
- Launching node explorer for node creates error and explorer closes [CORDA-2694]
- Transactions created in V3 cannot be verified in V4 if any of the state types were included in “depended upon” CorDapps which were not attached to the transaction [CORDA-2692]
- Reduce CorDapp scanning logging [CORDA-2690]
- Clean up verbose warning: *ProgressTracker has not been started* [CORDA-2689]
- Add a no-carpenter context [CORDA-2688]
- Improve CorDapp upgrade guidelines for migrating existing states on ledger (pre-V4) [CORDA-2684]
- SessionRejectException.UnknownClass trapped by flow hospital but no way to call dropSession-Init() [CORDA-2683]
- Repeated CordFormations can fail with ClassLoader exception. [CORDA-2676]
- Backwards compatibility break in serialisation engine when deserialising nullable fields [CORDA-2674]
- Simplify sample CorDapp projects. [CORDA-2672]
- Remove ExplorerSimulator from Node Explorer [CORDA-2671]
- Reintroduce pendingFlowsCount to the public API [CORDA-2669]
- Trader demo integration tests fails with jar not found exception [CORDA-2668]
- Fix Source ClassLoader for DJVM [CORDA-2667]
- Issue with simple transfer of ownable asset [CORDA-2665]
- Fix references to Docker images in docs [CORDA-2664]

- Add something to docsite the need for a common contracts Jar between OS/ENT and how it should be compiled against OS [CORDA-2656]
- Create document outlining CorDapp Upgrade guarantees [CORDA-2655]
- Fix DJVM CLI tool [CORDA-2654]
- Corda Service needs Thread Context ClassLoader [CORDA-2653]
- Useless migration error when finance workflow jar is not installed [CORDA-2651]
- Database connection pools leaking memory on every checkpoint [CORDA-2646]
- Exception swallowed when querying vault via RPC with bad page spec [CORDA-2645]
- Applying CordFormation and Cordapp Gradle plugins together includes Jolokia into the Cordapp. [CORDA-2642]
- Wrong folder ownership while trying to connect to Testnet using RC\* docker image [CORDA-2641]
- Provide a better error message on an incompatible implicit contract upgrade [CORDA-2633]
- uploadAttachment via shell can fail with unhelpful message if the result of the command is unsuccessful [CORDA-2632]
- Provide a better error msg when the notary type is misconfigured on the net params [CORDA-2629]
- Maybe tone down the level of panic when somebody types their SSH password in incorrectly... [CORDA-2621]
- Cannot complete transaction that has unknown states in the transaction history [CORDA-2615]
- Switch off the codepaths that disable the FinalityHandler [CORDA-2613]
- is our API documentation (what is stable and what isn't) correct? [CORDA-2610]
- Getting set up guide needs to be updated to reflect Java 8 fun and games [CORDA-2602]
- Not handle exception when Explorer tries to connect to inaccessible server [CORDA-2586]
- Errors received from peers can't be distinguished from local errors [CORDA-2572]
- Add *flow kill* command, deprecate *run killFlow* [CORDA-2569]
- Hash to signature constraints migration: add a config option that makes hash constraints breakable. [CORDA-2568]
- Deadlock between database and AppendOnlyPersistentMap [CORDA-2566]
- Docfix: Document custom cordapp configuration [CORDA-2560]
- Bootstrapper - option to include contracts to whitelist from signed jars [CORDA-2554]
- Explicit contract upgrade sample fails upon initiation (ClassNotFoundException) [CORDA-2550]
- IRS demo app missing demodate endpoint [CORDA-2535]
- Doc fix: Contract testing tutorial errors [CORDA-2528]
- Unclear error message when receiving state from node on higher version of signed cordapp [CORDA-2522]
- Terminating ssh connection to node results in stack trace being thrown to the console [CORDA-2519]
- Error propagating hash to signature constraints [CORDA-2515]
- Unable to import trusted attachment [CORDA-2512]
- Invalid node command line options not always gracefully handled [CORDA-2506]
- node.conf with rogue line results non-comprehensive error [CORDA-2505]

- Fix v4's inability to migrate V3 vault data [CORDA-2487]
- Vault Query fails to process states upon CorDapp Contract upgrade [CORDA-2486]
- Signature Constraints end-user documentation is limited [CORDA-2477]
- Docs update: document transition from the whitelist constraint to the sig constraint [CORDA-2465]
- The `ContractUpgradeWireTransaction` does not support the Signature Constraint [CORDA-2456]
- Intermittent *relation “hibernate\_sequence” does not exist* error when using Postgres [CORDA-2393]
- Implement package namespace ownership [CORDA-1947]
- Show explicit error message when new version of OS CorDapp contains schema changes [CORDA-1596]
- Dockerfile improvements and image size reduction [CORDA-2929]
- Update QPID Proton-J library to latest [CORDA-2856]
- Not handled exception when certificates directory is missing [CORDA-2786]
- The DJVM cannot sandbox instances of `Contract.verify(LedgerTransaction)` when testing CorDapps. [CORDA-2775]
- State machine logs an error prior to deciding to escalate to an error [CORDA-2757]
- Should Jolokia be included in the built jar files? [CORDA-2699]
- Transactions created in V3 cannot be verified in V4 if any of the state types were included in “depended upon” CorDapps which were not attached to the transaction [CORDA-2692]
- Prevent a node re-registering with the doorman if it did already and the node “state” has not been erased [CORDA-2647]
- The cert hierarchy diagram for C4 is the same as C3.0 but I thought we changed it between C3.1 and 3.2? [CORDA-2604]
- Windows build fails with *FileSystemException* in `TwoPartyTradeFlowTests` [CORDA-2363]
- *Cash.generateSpend* cannot be used twice to generate two cash moves in the same tx [CORDA-2162]
- FlowException thrown by `session.receive` is not propagated back to a counterparty
- invalid command line args for corda result in 0 exit code
- Windows build fails on `TwoPartyTradeFlowTests`
- C4 performance below C3, bring it back into parity
- Deserialisation of `ContractVerificationException` blows up trying to put null into non-null field
- Reference state test (R3T-1918) failing probably due to unconsumed linear state that was referenced.
- Signature constraint: Jarsigner verification allows removal of files from the archive.
- Node explorer bug revealed from within Demobench: serialisation failed error is shown
- Security: Fix vulnerability where an attacker can use `CustomSerializers` to alter the meaning of serialized data
- Node/RPC is broken after CorDapp upgrade
- RPC client disconnects shouldn't be a warning
- Hibernate logs warning and errors for some conditions we handle

## 1.2 Corda 4

Welcome to the Corda 4 release notes. Please read these carefully to understand what's new in this release and how the changes can help you. Just as prior releases have brought with them commitments to wire and API stability, Corda 4 comes with those same guarantees. States and apps valid in Corda 3 are transparently usable in Corda 4.

For app developers, we strongly recommend reading “[Upgrading apps to Corda 4](#)”. This covers the upgrade procedure, along with how you can adjust your app to opt-in to new features making your app more secure and easier to upgrade in future.

For node operators, we recommend reading “[Upgrading your node to Corda 4](#)”. The upgrade procedure is simple but it can't hurt to read the instructions anyway.

Additionally, be aware that the data model improvements are changes to the Corda consensus rules. To use apps that benefit from them, *all* nodes in a compatibility zone must be upgraded and the zone must be enforcing that upgrade. This may take time in large zones like the testnet. Please take this into account for your own schedule planning.

**Warning:** There is a bug in Corda 3.3 that causes problems when receiving a `FungibleState` created by Corda 4. There will shortly be a followup Corda 3.4 release that corrects this error. Interop between Corda 3 and Corda 4 will require that Corda 3 users are on the latest patchlevel release.

### 1.2.1 Changes for developers in Corda 4

#### Reference states

With Corda 4 we are introducing the concept of “reference input states”. These allow smart contracts to reference data from the ledger in a transaction without simultaneously updating it. They're useful not only for any kind of reference data such as rates, healthcare codes, geographical information etc, but for anywhere you might have used a `SELECT JOIN` in a SQL based app.

A reference input state is a `ContractState` which can be referred to in a transaction by the contracts of input and output states but, significantly, whose contract is not executed as part of the transaction verification process and is not consumed when the transaction is committed to the ledger. Rather, it is checked for “current-ness”. In other words, the contract logic isn't run for the referencing transaction only. Since they're normal states, if they do occur in the input or output positions, they can evolve on the ledger, modeling reference data in the real world.

#### Signature constraints

CorDapps built by the `corda-gradle-plugins` are now signed and sealed JAR files by default. This signing can be configured or disabled with the default certificate being the Corda development certificate.

When an app is signed, that automatically activates the use of signature constraints, which are an important part of the Corda security and upgrade plan. They allow states to express what contract logic governs them socially, as in “any contract JAR signed by a threshold of these N keys is suitable”, rather than just by hash or via zone whitelist rules, as in previous releases.

**We strongly recommend all apps be signed and use signature constraints going forward.**

Learn more about this new feature by reading the [Upgrading apps to Corda 4](#).

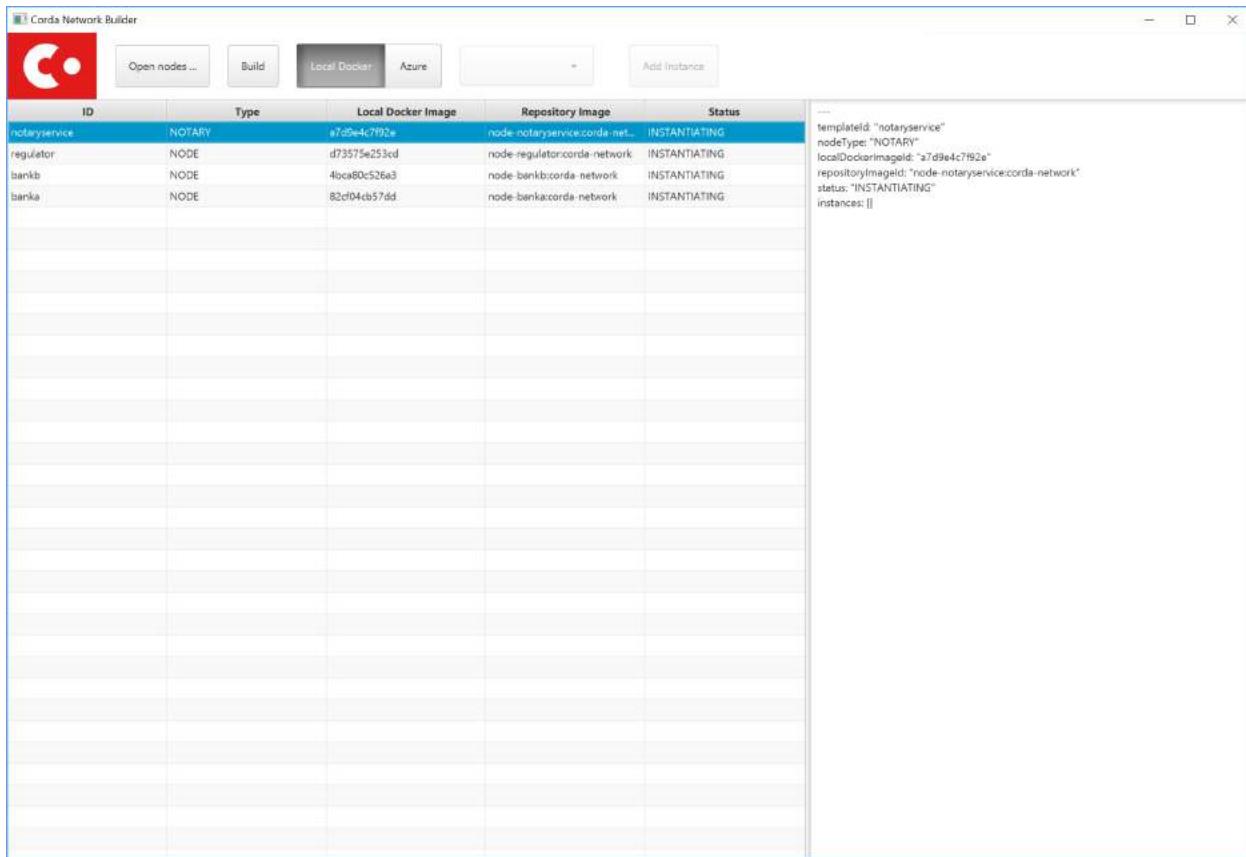
### State pointers

*State Pointers* formalize a recommended design pattern, in which states may refer to other states on the ledger by StateRef (a pair of transaction hash and output index that is sufficient to locate any information on the global ledger). State pointers work together with the reference states feature to make it easy for data to point to the latest version of any other piece of data, with the right version being automatically incorporated into transactions for you.

### New network builder tool

A new graphical tool for building test Corda networks has been added. It can build Docker images for local deployment and can also remotely control Microsoft Azure, to create a test network in the cloud.

Learn more on the [Corda Network Builder](#) page.



### JPA access in flows and services

Corda 3 provides the `jdbcTemplate` API on `FlowLogic` to give access to an active connection to your underlying database. It is fully intended that apps can store their own data in their own tables in the node database, so app-specific tables can be updated atomically with the ledger data itself. But JDBC is not always convenient, so in Corda 4 we are additionally exposing the *Java Persistence Architecture*, for object-relational mapping. The new `ServiceHub.withEntityManager` API lets you load and persist entity beans inside your flows and services.

Please do write apps that read and write directly to tables running alongside the node's own tables. Using SQL is a convenient and robust design pattern for accessing data on or off the ledger.

---

**Important:** Please do not attempt to write to tables starting with `node_` or `contract_` as those are maintained by the node. Additionally, the `node_` tables are private to Corda and should not be directly accessed at all. Tables starting with `contract_` are generated by apps and are designed to be queried by end users, GUIs, tools etc.

---

## Security upgrades

**Sealing.** Sealed JARs are a security upgrade that ensures JARs cannot define classes in each other's packages, thus ensuring Java's package-private visibility feature works. The Gradle plugins now seal your JARs by default.

**BelongsToContract annotation.** CorDapps are currently expected to verify that the right contract is named in each state object. This manual step is easy to miss, which would make the app less secure in a network where you trade with potentially malicious counterparties. The platform now handles this for you by allowing you to annotate states with which contract governs them. If states are inner classes of a contract class, this association is automatic. See [API: Contract Constraints](#) for more information.

**Two-sided FinalityFlow and SwapIdentitiesFlow.** The previous `FinalityFlow` API was insecure because nodes would accept any finalised transaction, outside of the context of a containing flow. This would allow transactions to be sent to a node bypassing things like business network membership checks. The same applies for the `SwapIdentitiesFlow` in the confidential-identities module. A new API has been introduced to allow secure use of this flow.

**Package namespace ownership.** Corda 4 allows app developers to register their keys and Java package namespaces with the zone operator. Any JAR that defines classes in these namespaces will have to be signed by those keys. This is an opt-in feature designed to eliminate potential confusion that could arise if a malicious developer created classes in other people's package namespaces (e.g. an attacker creating a state class called `com.megacorp.exampleapp.ExampleState`). Whilst Corda's attachments feature would stop the core ledger getting confused by this, tools and formats that connect to the node may not be designed to consider attachment hashes or signing keys, and rely more heavily on type names instead. Package namespace ownership allows tool developers to assume that if a class name appears to be owned by an organisation, then the semantics of that class actually *were* defined by that organisation, thus eliminating edge cases that might otherwise cause confusion.

## Network parameters in transactions

Transactions created under a Corda 4+ node will have the currently valid signed `NetworkParameters` file attached to each transaction. This will allow future introspection of states to ascertain what was the accepted global state of the network at the time they were notarised. Additionally, new signatures must be working with the current globally accepted parameters. The notary signing a transaction will check that it does indeed reference the current in-force network parameters, meaning that old (and superseded) network parameters can not be used to create new transactions.

## RPC upgrades

**AMQP/1.0** is now default serialization framework across all of Corda (checkpointing aside), swapping the RPC framework from using the older Kryo implementation. This means Corda open source and Enterprise editions are now RPC wire compatible and either client library can be used. We previously started using AMQP/1.0 for the peer to peer protocol in Corda 3.

**Class synthesis.** The RPC framework supports the “class carpenter” feature. Clients can now freely download and deserialise objects, such as contract states, for which the defining class files are absent from their classpath. Definitions for these classes will be synthesised on the fly from the binary schemas embedded in the messages. The resulting dynamically created objects can then be fed into any framework that uses reflection, such as XML formatters, JSON libraries, GUI construction toolkits, scripting engines and so on. This approach is how the [Blob Inspector](#) tool works -

it simply deserialises a message and then feeds the resulting synthetic class graph into a JSON or YAML serialisation framework.

Class synthesis will use interfaces that are implemented by the original objects if they are found on the classpath. This is designed to enable generic programming. For example, if your industry has standardised a thin Java API with interfaces that expose JavaBean style properties (get/is methods), then you can have that JAR on the classpath of your tool and cast the deserialised objects to those interfaces. In this way you can work with objects from apps you aren't aware of.

**SSL.** The Corda RPC infrastructure can now be configured to utilise SSL for additional security. The operator of a node wishing to enable this must of course generate and distribute a certificate in order for client applications to successfully connect. This is documented here [Using the client RPC API](#)

### Preview of the deterministic DJVM

It is important that all nodes that process a transaction always agree on whether it is valid or not. Because transaction types are defined using JVM byte code, this means that the execution of that byte code must be fully deterministic. Out of the box a standard JVM is not fully deterministic, thus we must make some modifications in order to satisfy our requirements.

This version of Corda introduces a standalone *Deterministic JVM*. It isn't yet integrated with the rest of the platform. It will eventually become a part of the node and enforce deterministic and secure execution of smart contract code, which is mobile and may propagate around the network without human intervention.

Currently, it is released as an evaluation version. We want to give developers the ability to start trying it out and get used to developing deterministic code under the set of constraints that we envision will be placed on contract code in the future. There are some instructions on how to get started with the DJVM command-line tool, which allows you to run code in a deterministic sandbox and inspect the byte code transformations that the DJVM applies to your code. Read more in “[Deterministic JVM](#)”.

### Configurable flow responders

In Corda 4 it is possible for flows in one app to subclass and take over flows from another. This allows you to create generic, shared flow logic that individual users can customise at pre-agreed points (protected methods). For example, a site-specific app could be developed that causes transaction details to be converted to a PDF and sent to a particular printer. This would be an inappropriate feature to put into shared business logic, but it makes perfect sense to put into a user-specific app they developed themselves.

If your flows could benefit from being extended in this way, read “[Configuring Responder Flows](#)” to learn more.

### Target/minimum versions

Applications can now specify a **target version** in their JAR manifest. The target version declares which version of the platform the app was tested against. By incrementing the target version, app developers can opt in to desirable changes that might otherwise not be entirely backwards compatible. For example in a future release when the deterministic JVM is integrated and enabled, apps will need to opt in to determinism by setting the target version to a high enough value.

Target versioning has a proven track record in both iOS and Android of enabling platforms to preserve strong backwards compatibility, whilst also moving forward with new features and bug fixes. We recommend that maintained applications always try and target the latest version of the platform. Setting a target version does not imply your app *requires* a node of that version, merely that it's been tested against that version and can handle any opt-in changes.

Applications may also specify a **minimum platform version**. If you try to install an app in a node that is too old to satisfy this requirement, the app won't be loaded. App developers can set their min platform version requirement if they start using new features and APIs.

## Dependency upgrades

We've raised the minimum JDK to 8u171, needed to get fixes for certain ZIP compression bugs.

We've upgraded to Kotlin 1.2.71 so your apps can now benefit from the new features in this language release.

We've upgraded to Gradle 4.10.1.

## 1.2.2 Changes for administrators in Corda 4

### Official Docker images

Corda 4 adds an *Official Corda Docker Image* for starting the node. It's based on Ubuntu and uses the Azul Zulu spin of Java 8. Other tools will have Docker images in future as well.

### Auto-acceptance for network parameters updates

Changes to the parameters of a compatibility zone require all nodes to opt in before a flag day.

Some changes are trivial and very unlikely to trigger any disagreement. We have added auto-acceptance for a subset of network parameters, negating the need for a node operator to manually run an accept command on every parameter update. This behaviour can be turned off via the node configuration. See *The network map*.

### Automatic error codes

Errors generated in Corda are now hashed to produce a unique error code that can be used to perform a lookup into a knowledge base. The lookup URL will be printed to the logs when an error occur. Here's an example:

```
[ERROR] 2018-12-19T17:18:39,199Z [main] internal.NodeStartupLogging.invoke -  
↳Exception during node startup: The name 'O=Wawrzek Test C4, L=London, C=GB' for  
↳identity doesn't match what's in the key store: O=Wawrzek Test C4, L=Ely, C=GB  
↳[errorCode=wuxa6f, moreInformationAt=https://errors.corda.net/OS/4.0/wuxa6f]
```

The hope is that common error conditions can quickly be resolved and opaque errors explained in a more user friendly format to facilitate faster debugging and trouble shooting.

At the moment, Stack Overflow is that knowledge base, with the error codes being converted to a URL that redirects either directly to the answer or to an appropriate search on Stack Overflow.

### Standardisation of command line argument handling

In Corda 4 we have ported the node and all our tools to use a new command line handling framework. Advantages for you:

- Improved, coloured help output.
- Common options have been standardised to use the same name and behaviour everywhere.
- All programs can now generate bash/zsh auto completion files.

You can learn more by reading our CLI user experience guidelines document.

## Liquibase for database schema upgrades

We have open sourced the Liquibase schema upgrade feature from Corda Enterprise. The node now uses Liquibase to bootstrap and update itself automatically. This is a transparent change with pre Corda 4 nodes seamlessly upgrading to operate as if they'd been bootstrapped in this way. This also applies to the finance CorDapp module.

---

**Important:** If you're upgrading a node from Corda 3 to Corda 4 and there is old data in the vault, this upgrade may take some time, depending on the number of unconsumed states in the vault.

---

## Ability to pre-validate configuration files

A new command has been added that lets you verify a config file is valid without starting up the rest of the node:

```
java -jar corda-4.0.jar validate-configuration
```

## Flow control for notaries

Notary clusters can now exert backpressure on clients, to stop them from being overloaded. Nodes will be ordered to back off if a notary is getting too busy, and app flows will pause to give time for the load spike to pass. This change is transparent to both developers and administrators.

## Retirement of non-elliptic Diffie-Hellman for TLS

The TLS\_DHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256 family of ciphers is retired from the list of allowed ciphers for TLS as it is a legacy cipher family not supported by all native SSL/TLS implementations. We anticipate that this will have no impact on any deployed configurations.

### 1.2.3 Miscellaneous changes

To learn more about smaller changes, please read the [Changelog](#).

Finally, we have added some new jokes. Thank you and good night!

## UPGRADING APPS TO CORDA 4

These notes provide instructions for upgrading your CorDapps from previous versions. Corda provides backwards compatibility for public, non-experimental APIs that have been committed to. A list can be found in the [Corda API](#) page.

This means that you can upgrade your node across versions *without recompiling or adjusting your CorDapps*. You just have to upgrade your node and restart.

However, there are usually new features and other opt-in changes that may improve the security, performance or usability of your application that are worth considering for any actively maintained software. This guide shows you how to upgrade your app to benefit from the new features in the latest release.

**Warning:** The sample apps found in the Corda repository and the Corda samples repository are not intended to be used in production. If you are using them you should re-namespace them to a package namespace you control, and sign/version them yourself.

### Contents

- *Upgrading apps to Corda 4*
  - *Step 1. Switch any RPC clients to use the new RPC library*
  - *Step 2. Adjust the version numbers in your Gradle build files*
  - *Step 3. Update your Gradle build file*
  - *Step 4. Remove any custom configuration from the node.conf*
  - *Step 5. Security: Upgrade your use of FinalityFlow*
    - \* *Upgrading a non-initiating flow*
    - \* *Upgrading an initiating flow*
  - *Step 6. Security: Upgrade your use of SwapIdentitiesFlow*
  - *Step 7. Possibly, adjust test code*
  - *Step 8. Security: Add BelongsToContract annotations*
  - *Step 9. Learn about signature constraints and JAR signing*
  - *Step 10. Security: Package namespace handling*
  - *Step 11. Consider adding extension points to your flows*
  - *Step 12. Possibly update vault state queries*

- Step 13. Explore other new features that may be useful
- Step 14. Possibly update your checked in quasar.jar

## 2.1 Step 1. Switch any RPC clients to use the new RPC library

Although the RPC API is backwards compatible with Corda 3, the RPC wire protocol isn't. Therefore RPC clients like web servers need to be updated in lockstep with the node to use the new version of the RPC library. Corda 4 delivers RPC wire stability and therefore in future you will be able to update the node and apps without updating RPC clients.

## 2.2 Step 2. Adjust the version numbers in your Gradle build files

Alter the versions you depend on in your Gradle file like so:

```
ext.corda_release_version = '4.1'  
ext.corda_gradle_plugins_version = '4.0.42'  
ext.kotlin_version = '1.2.71'  
ext.quasar_version = '0.7.10'
```

---

**Note:** You may wish to update your kotlinOptions to use language level 1.2, to benefit from the new features. Apps targeting Corda 4 may not at this time use Kotlin 1.3, as it was released too late in the development cycle for us to risk an upgrade. Sorry! Future work on app isolation will make it easier for apps to use newer Kotlin versions than the node itself uses.

---

You should also ensure you're using Gradle 4.10 (but not 5). If you use the Gradle wrapper, run:

```
./gradlew wrapper --gradle-version 4.10.3
```

Otherwise just upgrade your installed copy in the usual manner for your operating system.

## 2.3 Step 3. Update your Gradle build file

There are several adjustments that are beneficial to make to your Gradle build file, beyond simply incrementing the versions as described in step 1.

**Provide app metadata.** This is used by the Corda Gradle build plugin to populate your app JAR with useful information. It should look like this:

```
cordapp {  
    targetPlatformVersion 4  
    minimumPlatformVersion 4  
    contract {  
        name "MegaApp Contracts"  
        vendor "MegaCorp"  
        licence "A liberal, open source licence"  
        versionId 1  
    }  
    workflow {
```

(continues on next page)

(continued from previous page)

```

name "MegaApp flows"
vendor "MegaCorp"
licence "A really expensive proprietary licence"
versionId 1
}
}

```

**Important:** Watch out for the UK spelling of the word licence (with a c).

Name, vendor and licence can be set to any string you like, they don't have to be Corda identities.

Target versioning is a new concept introduced in Corda 4. Learn more by reading [Versioning](#). Setting a target version of 4 opts in to changes that might not be 100% backwards compatible, such as API semantics changes or disabling workarounds for bugs that may be in your apps, so by doing this you are promising that you have thoroughly tested your app on the new version. Using a high target version is a good idea because some features and improvements are only available to apps that opt in.

The minimum platform version is the platform version of the node that you require, so if you start using new APIs and features in Corda 4, you should set this to 4. Unfortunately Corda 3 and below do not know about this metadata and don't check it, so your app will still be loaded in such nodes and may exhibit undefined behaviour at runtime. However it's good to get in the habit of setting this properly for future releases.

**Note:** Whilst it's currently a convention that Corda releases have the platform version number as their major version i.e. Corda 3.3 implements platform version 3, this is not actually required and may in future not hold true. You should know the platform version of the node releases you want to target.

The new `versionId` number is a version code for **your** app, and is unrelated to Corda's own versions. It is used to informative purposes only. See "[App versioning with Signature Constraints](#)" for more information.

**Split your app into contract and workflow JARs.** The duplication between `contract` and `workflow` blocks exists because you should split your app into two separate JARs/modules, one that contains on-ledger validation code like states and contracts, and one for the rest (called by convention the "workflows" module although it can contain a lot more than just flows: services would also go here, for instance). For simplicity, here we use one JAR for both, but this is in general an anti-pattern and can result in your flow logic code being sent over the network to arbitrary third party peers, even though they don't need it.

In future, the version ID attached to the workflow JAR will also be used to help implement smoother upgrade and migration features. You may directly reference the gradle version number of your app when setting the CorDapp specific `versionId` identifiers if this follows the convention of always being a whole number starting from 1.

If you use the finance demo app, you should adjust your dependencies so you depend on the `finance-contracts` and `finance-workflows` artifacts from your own contract and workflow JAR respectively.

## 2.4 Step 4. Remove any custom configuration from the node.conf

CorDapps can no longer access custom configuration items in the `node.conf` file. Any custom CorDapp configuration should be added to a CorDapp configuration file. The Node's configuration will not be accessible. CorDapp configuration files should be placed in the `config` subdirectory of the Node's `cordapps` folder. The name of the file should match the name of the JAR of the CorDapp (eg; if your CorDapp is called `hello-0.1.jar` the configuration file needed would be `cordapps/config/hello-0.1.conf`).

If you are using the `extraConfig` of a node in the `deployNodes` Gradle task to populate custom configuration for testing, you will need to make the following change so that:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    node {
        name "O=Bank A,L=London,C=GB"c
        ...
        extraConfig = [ 'some.extra.config' : '12345' ]
    }
}
```

Would become:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    node {
        name "O=Bank A,L=London,C=GB"c
        ...
        projectCordapp {
            config "some.extra.config=12345"
        }
    }
}
```

See [CorDapp configuration files](#) for more information.

## 2.5 Step 5. Security: Upgrade your use of FinalityFlow

The previous `FinalityFlow` API is insecure. It doesn't have a receive flow, so requires counterparty nodes to accept any and all signed transactions that are sent to it, without checks. It is **highly** recommended that existing CorDapps migrate away to the new API, as otherwise things like business network membership checks won't be reliably enforced.

The flows that make use of `FinalityFlow` in a CorDapp can be classified in the following 2 basic categories:

- **non-initiating flows:** these are flows that finalise a transaction without the involvement of a counterpart flow at all.
- **initiating flows:** these are flows that initiate a counterpart (responder) flow.

There is a main difference between these 2 different categories, which is relevant to how the CorDapp can be upgraded. The second category of flows can be upgraded to use the new `FinalityFlow` in a backwards compatible way, which means the upgraded CorDapp can be deployed at the various nodes using a *rolling deployment*. On the other hand, the first category of flows cannot be upgraded to the new `FinalityFlow` in a backwards compatible way, so the changes to these flows need to be deployed simultaneously at all the nodes, using a *lockstep deployment*.

---

**Note:** A *lockstep deployment* is one, where all the involved nodes are stopped, upgraded to the new version of the CorDapp and then re-started. As a result, there can't be any nodes running different versions of the CorDapp at any time. A *rolling deployment* is one, where every node can be stopped, upgraded to the new version of the CorDapp and re-started independently and on its own pace. As a result, there can be nodes running different versions of the CorDapp and transact with each other successfully.

---

The upgrade is a three step process:

1. Change the flow that calls `FinalityFlow`.
2. Change or create the flow that will receive the finalised transaction.

3. Make sure your application's minimum and target version numbers are both set to 4 (see *Step 2. Adjust the version numbers in your Gradle build files*).

### 2.5.1 Upgrading a non-initiating flow

As an example, let's take a very simple flow that finalises a transaction without the involvement of a counterpart flow:

```
class SimpleFlowUsingOldApi(private val counterparty: Party) : FlowLogic
    <SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val stx = dummyTransactionWithParticipant(counterparty)
        return subFlow(FinalityFlow(stx))
    }
}
```

```
public static class SimpleFlowUsingOldApi extends FlowLogic<SignedTransaction> {
    private final Party counterparty;

    @Suspendable
    @Override
    public SignedTransaction call() throws FlowException {
        SignedTransaction stx = dummyTransactionWithParticipant(counterparty);
        return subFlow(new FinalityFlow(stx));
    }
}
```

To use the new API, this flow needs to be annotated with `InitiatingFlow` and a `FlowSession` to the participant(s) of the transaction must be passed to `FinalityFlow`:

```
// Notice how the flow *must* now be an initiating flow even when it wasn't before.
@InitiatingFlow
class SimpleFlowUsingNewApi(private val counterparty: Party) : FlowLogic
    <SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val stx = dummyTransactionWithParticipant(counterparty)
        // For each non-local participant in the transaction we must initiate a flow
        // session with them.
        val session = initiateFlow(counterparty)
        return subFlow(FinalityFlow(stx, session))
    }
}
```

```
// Notice how the flow *must* now be an initiating flow even when it wasn't before.
@InitiatingFlow
public static class SimpleFlowUsingNewApi extends FlowLogic<SignedTransaction> {
    private final Party counterparty;

    @Suspendable
    @Override
    public SignedTransaction call() throws FlowException {
        SignedTransaction stx = dummyTransactionWithParticipant(counterparty);
        // For each non-local participant in the transaction we must initiate a flow
        // session with them.
        FlowSession session = initiateFlow(counterparty);
    }
}
```

(continues on next page)

(continued from previous page)

```

    return subFlow(new FinalityFlow(stx, session));
}

```

If there are more than one transaction participants then a session to each one must be initiated, excluding the local party and the notary.

A responder flow has to be introduced, which will automatically run on the other participants' nodes, which will call `ReceiveFinalityFlow` to record the finalised transaction:

```

// All participants will run this flow to receive and record the finalised
// transaction into their vault.
@InitiatedBy(SimpleFlowUsingNewApi::class)
class SimpleNewResponderFlow(private val otherSide: FlowSession) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        subFlow(ReceiveFinalityFlow(otherSide))
    }
}

```

```

// All participants will run this flow to receive and record the finalised
// transaction into their vault.
@InitiatedBy(SimpleFlowUsingNewApi.class)
public static class SimpleNewResponderFlow extends FlowLogic<Void> {
    private final FlowSession otherSide;

    @Suspendable
    @Override
    public Void call() throws FlowException {
        subFlow(new ReceiveFinalityFlow(otherSide));
        return null;
    }
}

```

---

**Note:** As described above, all the nodes in your business network will need the new CorDapp, otherwise they won't know how to receive the transaction. **This includes nodes which previously didn't have the old CorDapp.** If a node is sent a transaction and it doesn't have the new CorDapp loaded then simply restart it with the CorDapp and the transaction will be recorded.

---

## 2.5.2 Upgrading an initiating flow

For flows which are already initiating counterpart flows then it's a matter of using the existing flow session. Note however, the new `FinalityFlow` is inlined and so the sequence of sends and receives between the two flows will change and will be incompatible with your current flows. You can use the flow version API to write your flows in a backwards compatible manner.

Here's what an upgraded initiating flow may look like:

```

// Assuming the previous version of the flow was 1 (the default if none is specified),
// we increment the version number to 2
// to allow for backwards compatibility with nodes running the old CorDapp.
@InitiatingFlow(version = 2)
class ExistingInitiatingFlow(private val counterparty: Party) : FlowLogic
    <SignedTransaction>() {
    @Suspendable

```

(continues on next page)

(continued from previous page)

```

override fun call(): SignedTransaction {
    val partiallySignedTx = dummyTransactionWithParticipant(counterparty)
    val session = initiateFlow(counterparty)
    val fullySignedTx = subFlow(CollectSignaturesFlow(partiallySignedTx, 
    ↪listOf(session)))
        // Determine which version of the flow that other side is using.
        return if (session.getCounterpartyFlowInfo().flowVersion == 1) {
            // Use the old API if the other side is using the previous version of the
↪flow.
            subFlow(FinalityFlow(fullySignedTx))
        } else {
            // Otherwise they're at least on version 2 and so we can send the
↪finalised transaction on the existing session.
            subFlow(FinalityFlow(fullySignedTx, session))
        }
    }
}

```

```

// Assuming the previous version of the flow was 1 (the default if none is specified),
↪ we increment the version number to 2
// to allow for backwards compatibility with nodes running the old CorDapp.
@InitiatingFlow(version = 2)
public static class ExistingInitiatingFlow extends FlowLogic<SignedTransaction> {
    private final Party counterparty;

    @Suspendable
    @Override
    public SignedTransaction call() throws FlowException {
        SignedTransaction partiallySignedTx = 
        ↪dummyTransactionWithParticipant(counterparty);
        FlowSession session = initiateFlow(counterparty);
        SignedTransaction fullySignedTx = subFlow(new
↪CollectSignaturesFlow(partiallySignedTx, singletonList(session))));
            // Determine which version of the flow that other side is using.
            if (session.getCounterpartyFlowInfo().getFlowVersion() == 1) {
                // Use the old API if the other side is using the previous version of the
↪flow.
                return subFlow(new FinalityFlow(fullySignedTx));
            } else {
                // Otherwise they're at least on version 2 and so we can send the
↪finalised transaction on the existing session.
                return subFlow(new FinalityFlow(fullySignedTx, session));
            }
        }
    }
}

```

For the responder flow, insert a call to `ReceiveFinalityFlow` at the location where it's expecting to receive the finalised transaction. If the initiator is written in a backwards compatible way then so must the responder.

```

// First we have to run the SignTransactionFlow, which will return a
↪SignedTransaction.
val txWeJustSigned = subFlow(object : SignTransactionFlow(otherSide) {
    @Suspendable
    override fun checkTransaction(stx: SignedTransaction) {
        // Implement responder flow transaction checks here
    }
})

```

(continues on next page)

(continued from previous page)

```

if (otherSide.getCounterpartyFlowInfo().flowVersion >= 2) {
    // The other side is not using the old CorDapp so call ReceiveFinalityFlow to
    // record the finalised transaction.
    // If SignTransactionFlow is used then we can verify the transaction we receive
    // for recording is the same one
    // that was just signed.
    subFlow(ReceiveFinalityFlow(otherSide, expectedTxId = txWeJustSigned.id))
} else {
    // Otherwise the other side is running the old CorDapp and so we don't need to do
    // anything further. The node
    // will automatically record the finalised transaction using the old insecure
    // mechanism.
}

```

```

// First we have to run the SignTransactionFlow, which will return a
// SignedTransaction.
SignedTransaction txWeJustSigned = subFlow(new SignTransactionFlow(otherSide) {
    @Suspendable
    @Override
    protected void checkTransaction(@NotNull SignedTransaction stx) throws
    FlowException {
        // Implement responder flow transaction checks here
    }
});

if (otherSide.getCounterpartyFlowInfo().getFlowVersion() >= 2) {
    // The other side is not using the old CorDapp so call ReceiveFinalityFlow to
    // record the finalised transaction.
    // If SignTransactionFlow is used then we can verify the transaction we receive
    // for recording is the same one
    // that was just signed by passing the transaction id to ReceiveFinalityFlow.
    subFlow(new ReceiveFinalityFlow(otherSide, txWeJustSigned.getId()));
} else {
    // Otherwise the other side is running the old CorDapp and so we don't need to do
    // anything further. The node
    // will automatically record the finalised transaction using the old insecure
    // mechanism.
}

```

You may already be using `waitForLedgerCommit` in your responder flow for the finalised transaction to appear in the local node's vault. Now that it's calling `ReceiveFinalityFlow`, which effectively does the same thing, this is no longer necessary. The call to `waitForLedgerCommit` should be removed.

## 2.6 Step 6. Security: Upgrade your use of SwapIdentitiesFlow

The `Confidential identities` API is experimental in Corda 3 and remains so in Corda 4. In this release, the `SwapIdentitiesFlow` has been adjusted in the same way as `FinalityFlow` above, to close problems with confidential identities being injectable into a node outside of other flow context. Old code will still work, but it is recommended to adjust your call sites so a session is passed into the `SwapIdentitiesFlow`.

## 2.7 Step 7. Possibly, adjust test code

`MockNodeParameters` and functions creating it no longer use a lambda expecting a `NodeConfiguration` object. Use a `MockNetworkConfigOverrides` object instead. This is an API change we regret, but unfortunately in Corda 3 we accidentally exposed large amounts of the node internal code through this one API entry point. We have now insulated the test API from node internals and reduced the exposure.

If you are constructing a `MockServices` for testing contracts, and your contract uses the `Cash` contract from the finance app, you now need to explicitly add `net.corda.finance.contracts` to the list of `cordappPackages`. This is a part of the work to disentangle the finance app (which is really a demo app) from the Corda internals. Example:

```
val ledgerServices = MockServices(
    listOf("net.corda.examples.obligation", "net.corda.testing.contracts"),
    initialIdentity = TestIdentity(CordaX500Name("TestIdentity", "", "GB")),
    identityService = makeTestIdentityService()
)
```

```
MockServices ledgerServices = new MockServices(
    Arrays.asList("net.corda.examples.obligation", "net.corda.testing.contracts"),
    new TestIdentity(new CordaX500Name("TestIdentity", "", "GB")),
    makeTestIdentityService()
);
```

becomes:

```
val ledgerServices = MockServices(
    listOf("net.corda.examples.obligation", "net.corda.testing.contracts", "net.corda.
        ↪finance.contracts"),
    initialIdentity = TestIdentity(CordaX500Name("TestIdentity", "", "GB")),
    identityService = makeTestIdentityService()
)
```

```
MockServices ledgerServices = new MockServices(
    Arrays.asList("net.corda.examples.obligation", "net.corda.testing.contracts",
        ↪"net.corda.finance.contracts"),
    new TestIdentity(new CordaX500Name("TestIdentity", "", "GB")),
    makeTestIdentityService()
);
```

You may need to use the new `TestCordapp` API when testing with the node driver or mock network, especially if you decide to stick with the pre-Corda 4 `FinalityFlow` API. The previous way of pulling in CorDapps into your tests (i.e. via using the `cordappPackages` parameter) does not honour CorDapp versioning. The new API `TestCordapp.findCordapp()` discovers the CorDapps that contain the provided packages scanning the classpath, so you have to ensure that the classpath the tests are running under contains either the `CorDapp.jar` or (if using Gradle) the relevant Gradle sub-project. In the first case, the versioning information in the `CorDapp.jar` file will be maintained. In the second case, the versioning information will be retrieved from the Gradle `cordapp` task. For example, if you are using `MockNetwork` for your tests, the following code:

```
val mockNetwork = MockNetwork(
    cordappPackages = listOf("net.corda.examples.obligation", "net.corda.finance.
        ↪contracts"),
    notarySpecs = listOf(MockNetworkNotarySpec(notary))
)
```

```
MockNetwork mockNetwork = new MockNetwork(
    Arrays.asList("net.corda.examples.obligation", "net.corda.finance.contracts"),
    new MockNetworkParameters().withNotarySpecs(Arrays.asList(new
        ↪MockNetworkNotarySpec(notary)))
);
```

would need to be transformed into:

```
val mockNetwork = MockNetwork(
    MockNetworkParameters(
        cordappsForAllNodes = listOf(
            TestCordapp.findCordapp("net.corda.examples.obligation.contracts"),
            TestCordapp.findCordapp("net.corda.examples.obligation.flows")
        ),
        notarySpecs = listOf(MockNetworkNotarySpec(notary))
    )
)
```

```
MockNetwork mockNetwork = new MockNetwork(
    new MockNetworkParameters(
        Arrays.asList(
            TestCordapp.findCordapp("net.corda.examples.obligation.contracts"),
            TestCordapp.findCordapp("net.corda.examples.obligation.flows")
        )
    ).withNotarySpecs(Arrays.asList(new MockNetworkNotarySpec(notary)))
);
```

Note that every package should exist in only one CorDapp, otherwise the discovery process won't be able to determine which one to use and you will most probably see an exception telling you There is more than one CorDapp containing the package. For instance, if you have 2 CorDapps containing the packages `net.corda.examples.obligation.contracts` and `net.corda.examples.obligation.flows`, you will get this error if you specify the package `net.corda.examples.obligation`.

---

**Note:** If you have any CorDapp code (e.g. flows/contracts/states) that is only used by the tests and located in the same test module, it won't be discovered now. You will need to move them in the main module of one of your CorDapps or create a new, separate CorDapp for them, in case you don't want this code to live inside your production CorDapps.

---

## 2.8 Step 8. Security: Add BelongsToContract annotations

In versions of the platform prior to v4, it was the responsibility of contract and flow logic to ensure that `TransactionState` objects contained the correct class name of the expected contract class. If these checks were omitted, it would be possible for a malicious counterparty to construct a transaction containing e.g. a cash state governed by a commercial paper contract. The contract would see that there were no commercial paper states in a transaction and do nothing, i.e. accept.

In Corda 4 the platform takes over this responsibility from the app, if the app has a target version of 4 or higher. A state is expected to be governed by a contract that is either:

1. The outer class of the state class, if the state is an inner class of a contract. This is a common design pattern.
2. Annotated with `@BelongsToContract` which specifies the contract class explicitly.

Learn more by reading [Contract/State Agreement](#). If an app targets Corda 3 or lower (i.e. does not specify a target version), states that point to contracts outside their package will trigger a log warning but validation will proceed.

## 2.9 Step 9. Learn about signature constraints and JAR signing

*Signature Constraints* are a new data model feature introduced in Corda 4. They make it much easier to deploy application upgrades smoothly and in a decentralised manner. Signature constraints are the new default mode for CorDapps, and the act of upgrading your app to use the version 4 Gradle plugins will result in your app being automatically signed, and new states automatically using new signature constraints selected automatically based on these signing keys.

You can read more about signature constraints and what they do in [API: Contract Constraints](#). The TransactionBuilder class will automatically use them if your application JAR is signed. **We recommend all JARs are signed.** To learn how to sign your JAR files, read [Signing the CorDapp JAR](#). In dev mode, all JARs are signed by developer certificates. If a JAR that was signed with developer certificates is deployed to a production node, the node will refuse to start. Therefore to deploy apps built for Corda 4 to production you will need to generate signing keys and integrate them with the build process.

---

**Note:** Please read the [CorDapp constraints migration](#) guide to understand how to upgrade CorDapps to use Corda 4 signature constraints and consume existing states on ledger issued with older constraint types (e.g. Corda 3.x states issued with **hash** or **CZ whitelisted** constraints).

---

## 2.10 Step 10. Security: Package namespace handling

Almost no apps will be affected by these changes, but they're important to know about.

There are two improvements to how Java package protection is handled in Corda 4:

1. Package sealing
2. Package namespace ownership

**Sealing.** App isolation has been improved. Version 4 of the finance CorDapps (`corda-finance-contracts.jar`, `corda-finance-workflows.jar`) is now built as a set of sealed and signed JAR files. This means classes in your own CorDapps cannot be placed under the following package namespace: `net.corda.finance`

In the unlikely event that you were injecting code into `net.corda.finance.*` package namespaces from your own apps, you will need to move them into a new package, e.g. `net/corda/finance/flows/MyClass.java` can be moved to `com/company/corda/finance/flows/MyClass.java`. As a consequence your classes are no longer able to access non-public members of finance CorDapp classes.

When signing your JARs for Corda 4, your own apps will also become sealed, meaning other JARs cannot place classes into your own packages. This is a security upgrade that ensures package-private visibility in Java code works correctly. If other apps could define classes in your own packages, they could call package-private methods, which may not be expected by the developers.

**Namespace ownership.** This part is only relevant if you are joining a production compatibility zone. You may wish to contact your zone operator and request ownership of your root package namespaces (e.g. `com.megacorp.*`), with the signing keys you will be using to sign your app JARs. The zone operator can then add your signing key to the network parameters, and prevent attackers defining types in your own package namespaces. Whilst this feature is optional and not strictly required, it may be helpful to block attacks at the boundaries of a Corda based application where type names may be taken “as read”. You can learn more about this feature and the motivation for it by reading “design/data-model-upgrades/package-namespace-ownership”.

## 2.11 Step 11. Consider adding extension points to your flows

In Corda 4 it is possible for flows in one app to subclass and take over flows from another. This allows you to create generic, shared flow logic that individual users can customise at pre-agreed points (protected methods). For example, a site-specific app could be developed that causes transaction details to be converted to a PDF and sent to a particular printer. This would be an inappropriate feature to put into shared business logic, but it makes perfect sense to put into a user-specific app they developed themselves.

If your flows could benefit from being extended in this way, read “[Configuring Responder Flows](#)” to learn more.

## 2.12 Step 12. Possibly update vault state queries

In Corda 4 queries made on a node’s vault can filter by the relevancy of those states to the node. As this functionality does not exist in Corda 3, apps will continue to receive all states in any vault queries. However, it may make sense to migrate queries expecting just those states relevant to the node in question to query for only relevant states. See [API: Vault Query](#) for more details on how to do this. Not doing this may result in queries returning more states than expected if the node is using observer functionality (see “[Observer nodes](#)”).

## 2.13 Step 13. Explore other new features that may be useful

Corda 4 adds several new APIs that help you build applications. Why not explore:

- The [new EntityManager API](#) for using JPA inside your flows and services.
- [Reference States](#), that let you use an input state without consuming it.
- [State Pointers](#), that make it easier to ‘point’ to one state from another and follow the latest version of a linear state.

Please also read the [CorDapp Upgradeability Guarantees](#) associated with CorDapp upgrading.

## 2.14 Step 14. Possibly update your checked in quasar.jar

If your project is based on one of the official cordapp templates, it is likely you have a `lib/quasar.jar` checked in. It is worth noting that you only use this if you use the JUnit runner in IntelliJ. In the latest release of the cordapp templates, this directory has been removed.

You have some choices here:

- Upgrade your `quasar.jar` to 0.7.10
- Delete your `lib` directory and switch to using the Gradle test runner

Instructions for both options can be found in [Running tests in IntelliJ](#).

## UPGRADING YOUR NODE TO CORDA 4

Corda releases strive to be backwards compatible, so upgrading a node is fairly straightforward and should not require changes to applications. It consists of the following steps:

1. Drain the node.
2. Make a backup of your node directories and/or database.
3. Replace the `corda.jar` file with the new version.
4. Start up the node. This step may incur a delay whilst any needed database migrations are applied.
5. Undrain it to re-enable processing of new inbound flows.

The protocol is designed to tolerate node outages, so during the upgrade process peers on the network will wait for your node to come back.

### 3.1 Step 1. Drain the node

Before a node or application on it can be upgraded, the node must be put in *Draining mode*. This brings the currently running *Flows* to a smooth halt such that existing work is finished and new work is queuing up rather than being processed.

Draining flows is a key task for node administrators to perform. It exists to simplify applications by ensuring apps don't have to be able to migrate workflows from any arbitrary point to other arbitrary points, a task that would rapidly become infeasible as workflow and protocol complexity increases.

To drain the node, run the `gracefulShutdown` command. This will wait for the node to drain and then shut down the node when the drain is complete.

**Warning:** The length of time a node takes to drain depends on both how your applications are designed, and whether any apps are currently talking to network peers that are offline or slow to respond. It is thus hard to give guidance on how long a drain should take, but in an environment with well written apps and in which your counterparties are online, drains may need only a few seconds.

### 3.2 Step 2. Make a backup of your node directories and/or database

It's always a good idea to make a backup of your data before upgrading any server. This will make it easy to roll back if there's a problem. You can simply make a copy of the node's data directory to enable this. If you use an external non-H2 database please consult your database user guide to learn how to make backups.

We provide some *backup recommendations* if you'd like more detail.

### 3.3 Step 3. Upgrade the node database to Corda 3.2 or later

Ensure your node is running Corda 3.2 or later. Corda 3.2 required a database table name change and column type change in PostgreSQL. These changes need to be applied to the database before upgrading to Corda 4.0. Refer to Corda 3.2 release notes for further information.

### 3.4 Step 4. Replace `corda.jar` with the new version

Download the latest version of Corda from [our Artifactory site](#). Make sure it's available on your path, and that you've read the [\*Release notes\*](#), in particular to discover what version of Java this node requires.

---

**Important:** Corda 4 requires Java 8u171 or any higher Java 8 patchlevel. Java 9+ is not currently supported.

---

### 3.5 Step 5. Start up the node

Start the node in the usual manner you have selected. The node will perform any automatic data migrations required, which may take some time. If the migration process is interrupted it can be continued simply by starting the node again, without harm.

### 3.6 Step 6. Undrain the node

You may now do any checks that you wish to perform, read the logs, and so on. When you are ready, use this command at the shell:

```
run setFlowsDrainingModeEnabled enabled: false
```

Your upgrade is complete.

## CORDA API

The following are the core APIs that are used in the development of CorDapps:

### 4.1 API: States

---

**Note:** Before reading this page, you should be familiar with the key concepts of *States*.

---

#### Contents

- *API: States*
  - *ContractState*
  - *ContractState sub-interfaces*
    - \* *LinearState*
    - \* *OwnableState*
      - *FungibleState*
    - \* *Other interfaces*
  - *User-defined fields*
  - *The vault*
  - *TransactionState*
  - *Reference States*
  - *State Pointers*

#### 4.1.1 ContractState

In Corda, states are instances of classes that implement `ContractState`. The `ContractState` interface is defined as follows:

```
/**  
 * A contract state (or just "state") contains opaque data used by a contract program.  
 * It can be thought of as a disk
```

(continues on next page)

(continued from previous page)

```

* file that the program can use to persist data across transactions. States are u
↳ immutable: once created they are never
* updated, instead, any changes must generate a new successor state. States can be u
↳ updated (consumed) only once: the
* notary is responsible for ensuring there is no "double spending" by only signing a u
↳ transaction if the input states
* are all free.
*/
@KeepForDJVM
@CordaSerializable
interface ContractState {
    /**
     * A _participant_ is any party that should be notified when the state is created u
↳ or consumed.
     *
     * The list of participants is required for certain types of transactions. For u
↳ example, when changing the notary
     * for this state, every participant has to be involved and approve the u
↳ transaction
     * so that they receive the updated state, and don't end up in a situation where u
↳ they can no longer use a state
     * they possess, since someone consumed that state during the notary change u
↳ process.
     *
     * The participants list should normally be derived from the contents of the u
↳ state.
     */
    val participants: List<AbstractParty>
}

```

`ContractState` has a single field, `participants`. `participants` is a `List` of the `AbstractParty` that are considered to have a stake in the state. Among other things, the `participants` will:

- Usually store the state in their vault (see below)
- Need to sign any notary-change and contract-upgrade transactions involving this state
- Receive any finalised transactions involving this state as part of `FinalityFlow`/`ReceiveFinalityFlow`

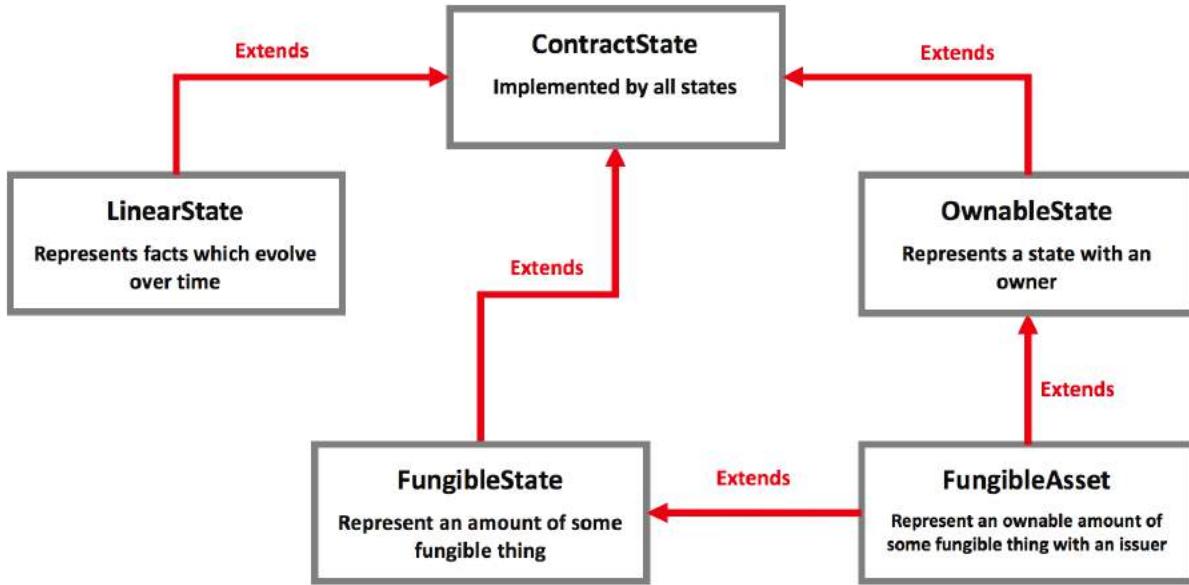
### 4.1.2 ContractState sub-interfaces

The behaviour of the state can be further customised by implementing sub-interfaces of `ContractState`. The two most common sub-interfaces are:

- `LinearState`
- `OwnableState`

`LinearState` models shared facts for which there is only one current version at any point in time. `LinearState` states evolve in a straight line by superseding themselves. On the other hand, `OwnableState` is meant to represent assets that can be freely split and merged over time. Cash is a good example of an `OwnableState` - two existing \$5 cash states can be combined into a single \$10 cash state, or split into five \$1 cash states. With `OwnableState`, its the total amount held that is important, rather than the actual units held.

We can picture the hierarchy as follows:



## LinearState

The `LinearState` interface is defined as follows:

```

/**
 * A state that evolves by superseding itself, all of which share the common "linearId".
 */
/**
 * This simplifies the job of tracking the current version of certain types of state
 * in e.g. a vault.
 */
@KeepForDJVM
interface LinearState : ContractState {
    /**
     * Unique id shared by all LinearState states throughout history within the
     * vaults of all parties.
     * Verify methods should check that one input and one output share the id in a
     * transaction,
     * except at issuance/termination.
    */
    val linearId: UniqueIdentifier
}
  
```

Remember that in Corda, states are immutable and can't be updated directly. Instead, we represent an evolving fact as a sequence of `LinearState` states that share the same `linearId` and represent an audit trail for the lifecycle of the fact over time.

When we want to extend a `LinearState` chain (i.e. a sequence of states sharing a `linearId`), we:

- Use the `linearId` to extract the latest state in the chain from the vault
- Create a new state that has the same `linearId`
- Create a transaction with:
  - The current latest state in the chain as an input

- The newly-created state as an output

The new state will now become the latest state in the chain, representing the new current state of the agreement.

`linearId` is of type `UniqueIdentifier`, which is a combination of:

- A Java `UUID` representing a globally unique 128 bit random number
- An optional external-reference string for referencing the state in external systems

### OwnableState

The `OwnableState` interface is defined as follows:

```
/**  
 * Return structure for [OwnableState.withNewOwner]  
 */  
@KeepForDJVM  
data class CommandAndState(val command: CommandData, val ownableState: OwnableState)  
  
/**  
 * A contract state that can have a single owner.  
 */  
@KeepForDJVM  
interface OwnableState : ContractState {  
    /** There must be a MoveCommand signed by this key to claim the amount. */  
    val owner: AbstractParty  
  
    /** Copies the underlying data structure, replacing the owner field with this new  
     * value and leaving the rest alone. */  
    fun withNewOwner(newOwner: AbstractParty): CommandAndState  
}
```

Where:

- `owner` is the `PublicKey` of the asset's owner
- `withNewOwner(newOwner: AbstractParty)` creates an copy of the state with a new owner

Because `OwnableState` models fungible assets that can be merged and split over time, `OwnableState` instances do not have a `linearId`. \$5 of cash created by one transaction is considered to be identical to \$5 of cash produced by another transaction.

### FungibleState

`FungibleState<T>` is an interface to represent things which are fungible, this means that there is an expectation that these things can be split and merged. That's the only assumption made by this interface. This interface should be implemented if you want to represent fractional ownership in a thing, or if you have many things. Examples:

- There is only one Mona Lisa which you wish to issue 100 tokens, each representing a 1% interest in the Mona Lisa
- A company issues 1000 shares with a nominal value of 1, in one batch of 1000. This means the single batch of 1000 shares could be split up into 1000 units of 1 share.

The interface is defined as follows:

```
@KeepForDJKM
interface FungibleState<T : Any> : ContractState {
    /**
     * Amount represents a positive quantity of some token which can be cash, tokens, stock, agreements, or generally anything else that's quantifiable with integer quantities. See [Amount] for more details.
     */
    val amount: Amount<T>
}
```

As seen, the interface takes a type parameter `T` that represents the fungible thing in question. This should describe the basic type of the asset e.g. GBP, USD, oil, shares in company `<X>`, etc. and any additional metadata (issuer, grade, class, etc.). An upper-bound is not specified for `T` to ensure flexibility. Typically, a class would be provided that implements `TokenizableAssetInfo` so the thing can be easily added and subtracted using the `Amount` class.

This interface has been added in addition to `FungibleAsset` to provide some additional flexibility which `FungibleAsset` lacks, in particular:

- `FungibleAsset` defines an amount property of type `Amount<Issued<T>>`, therefore there is an assumption that all fungible things are issued by a single well known party but this is not always the case.
- `FungibleAsset` implements `OwnableState`, as such there is an assumption that all fungible things are ownable.

## Other interfaces

You can also customize your state by implementing the following interfaces:

- `QueryableState`, which allows the state to be queried in the node's database using custom attributes (see [API: Persistence](#))
- `SchedulableState`, which allows us to schedule future actions for the state (e.g. a coupon payment on a bond) (see [Event scheduling](#))

### 4.1.3 User-defined fields

Beyond implementing `ContractState` or a sub-interface, a state is allowed to have any number of additional fields and methods. For example, here is the relatively complex definition for a state representing cash:

```
/** A state representing a cash claim against some party. */
@BelongsToContract(Cash::class)
data class State(
    override val amount: Amount<Issued<Currency>>,
    /**
     * There must be a MoveCommand signed by this key to claim the amount.
     */
    override val owner: AbstractParty
) : FungibleAsset<Currency>, QueryableState {
    constructor(deposit: PartyAndReference, amount: Amount<Currency>, owner: AbstractParty)
        : this(Amount(amount.quantity, Issued(deposit, amount.token)), owner)

    override val exitKeys = setOf(owner.owningKey, amount.token.issuer.party)
    override val participants = listOf(owner)
```

(continues on next page)

(continued from previous page)

```

override fun withNewOwnerAndAmount(newAmount: Amount<Issued<Currency>>, ↵
→newOwner: AbstractParty): FungibleAsset<Currency>
    = copy(amount = amount.copy(newAmount.quantity), owner = newOwner)

override fun toString() = "${Emoji.bagOfCash}Cash($amount at ${amount.token.↵
→issuer} owned by $owner)"

override fun withNewOwner(newOwner: AbstractParty) = CommandAndState(Command.↵
→Move(), copy(owner = newOwner))
    infix fun ownedBy(owner: AbstractParty) = copy(owner = owner)
    infix fun issuedBy(party: AbstractParty) = copy(amount = Amount(amount.↵
→quantity, amount.token.copy(issuer = amount.token.issuer.copy(party = party))))
        infix fun issuedBy(deposit: PartyAndReference) = copy(amount = Amount(amount.↵
→quantity, amount.token.copy(issuer = deposit)))
        infix fun withDeposit(deposit: PartyAndReference): Cash.State = copy(amount = ↵
→amount.copy(token = amount.token.copy(issuer = deposit)))

/** Object Relational Mapping support. */
override fun generateMappedObject(schema: MappedSchema): PersistentState {
    return when (schema) {
        is CashSchemaV1 -> CashSchemaV1.PersistentCashState(
            owner = this.owner,
            pennies = this.amount.quantity,
            currency = this.amount.token.product.currencyCode,
            issuerPartyHash = this.amount.token.issuer.party.owningKey.
→toStringShort(),
            issuerRef = this.amount.token.issuer.reference.bytes
        )
        /** Additional schema mappings would be added here (eg. CashSchemaV2, ↵
→CashSchemaV3, ...) else -> throw IllegalArgumentException("Unrecognised schema $schema")
    }
}

/** Object Relational Mapping support. */
override fun supportedSchemas(): Iterable<MappedSchema> = listOf(CashSchemaV1)
/** Additional used schemas would be added here (eg. CashSchemaV2, ↵
→CashSchemaV3, ...)
}

```

#### 4.1.4 The vault

Whenever a node records a new transaction, it also decides whether it should store each of the transaction's output states in its vault. The default vault implementation makes the decision based on the following rules:

- If the state is an `OwnableState`, the vault will store the state if the node is the state's owner
- Otherwise, the vault will store the state if it is one of the participants

States that are not considered relevant are not stored in the node's vault. However, the node will still store the transactions that created the states in its transaction storage.

#### 4.1.5 TransactionState

When a `ContractState` is added to a `TransactionBuilder`, it is wrapped in a `TransactionState`:

```

typealias ContractClassName = String

/**
 * A wrapper for [ContractState] containing additional platform-level state
 * information and contract information.
 * This is the definitive state that is stored on the ledger and used in transaction
 * outputs.
 */
@CordaSerializable
data class TransactionState<out T : ContractState> @JvmOverloads constructor(
    /** The custom contract state */
    val data: T,
    /**
     * The contract class name that will verify this state that will be created
     * via reflection.
     * The attachment containing this class will be automatically added to the
     * transaction at transaction creation
     * time.
     *
     * Currently these are loaded from the classpath of the node which includes
     * the cordapp directory - at some
     * point these will also be loaded and run from the attachment store directly,
     * allowing contracts to be
     * sent across, and run, from the network from within a sandbox environment.
     */
    // TODO: Implement the contract sandbox loading of the contract attachments
    val contract: ContractClassName = requireNotNull(data.
    requiredContractClassName) {
        //TODO: add link to docsite page, when there is one.
        """
        Unable to infer Contract class name because state class ${data::class.java.name}
        is not annotated with
        @BelongsToContract, and does not have an enclosing class which implements
        Contract. Either annotate ${data::class.java.name}
        with @BelongsToContract, or supply an explicit contract parameter to
        TransactionState().
        """.trimIndent().replace('\n', ' ')
    },
    /** Identity of the notary that ensures the state is not used as an input to
     * a transaction more than once */
    val notary: Party,
    /**
     * All contract states may be _encumbered_ by up to one other state.
     *
     * The encumbrance state, if present, forces additional controls over the
     * encumbered state, since the platform checks
     * that the encumbrance state is present as an input in the same transaction
     * that consumes the encumbered state, and
     * the contract code and rules of the encumbrance state will also be verified
     * during the execution of the transaction.
     * For example, a cash contract state could be encumbered with a time-lock
     * contract state; the cash 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 another by index, and the referred
     * encumbrance state
    */
}

```

(continues on next page)

(continued from previous page)

```

    * is an output state in a particular position on the same transaction that
    ↪created the encumbered state. An alternative
        * implementation would be encumbering by reference to a [StateRef], which
    ↪would allow the specification of encumbrance
            * by a state created in a prior transaction.
            *
            * Note that an encumbered state that is being consumed must have its
    ↪encumbrance consumed in the same transaction,
            * otherwise the transaction is not valid.
            */
val encumbrance: Int? = null,
/***
 * A validator for the contract attachments on the transaction.
***/
val constraint: AttachmentConstraint = AutomaticPlaceholderConstraint) {

private companion object {
    val logger = loggerFor<TransactionState<*>>()
}

init {
    when {
        data.requiredContractClassName == null -> logger.warn(
            """
                State class ${data::class.java.name} is not annotated with @BelongsToContract,
                and does not have an enclosing class which implements Contract. Annotate ${
        ↪{data::class.java.simpleName}
            with @BelongsToContract(${contract.split("\\\\\$").last()}.class) to remove
    ↪this warning.
            """.trimIndent().replace('\n', ' ')
        )
        data.requiredContractClassName != contract -> logger.warn(
            """
                State class ${data::class.java.name} belongs to contract ${data.
    ↪requiredContractClassName},
                    but is bundled with contract $contract in TransactionState. Annotate ${
        ↪{data::class.java.simpleName}
                    with @BelongsToContract(${contract.split("\\\\\$").last()}.class) to remove
    ↪this warning.
            """.trimIndent().replace('\n', ' ')
        )
    }
}
}

```

Where:

- **data** is the state to be stored on-ledger
- **contract** is the contract governing evolutions of this state
- **notary** is the notary service for this state
- **encumbrance** points to another state that must also appear as an input to any transaction consuming this state
- **constraint** is a constraint on which contract-code attachments can be used with this state

## 4.1.6 Reference States

A reference input state is a `ContractState` which can be referred to in a transaction by the contracts of input and output states but whose contract is not executed as part of the transaction verification process. Furthermore, reference states are not consumed when the transaction is committed to the ledger but they are checked for “current-ness”. In other words, the contract logic isn’t run for the referencing transaction only. It’s still a normal state when it occurs in an input or output position.

Reference data states enable many parties to reuse the same state in their transactions as reference data whilst still allowing the reference data state owner the capability to update the state. A standard example would be the creation of financial instrument reference data and the use of such reference data by parties holding the related financial instruments.

Just like regular input states, the chain of provenance for reference states is resolved and all dependency transactions verified. This is because users of reference data must be satisfied that the data they are referring to is valid as per the rules of the contract which governs it and that all previous participants of the state assented to updates of it.

### Known limitations:

*Notary change:* It is likely the case that users of reference states do not have permission to change the notary assigned to a reference state. Even if users *did* have this permission the result would likely be a bunch of notary change races. As such, if a reference state is added to a transaction which is assigned to a different notary to the input and output states then all those inputs and outputs must be moved to the notary which the reference state uses.

If two or more reference states assigned to different notaries are added to a transaction then it follows that this transaction cannot be committed to the ledger. This would also be the case for transactions not containing reference states. There is an additional complication for transactions including reference states; it is however, unlikely that the party using the reference states has the authority to change the notary for the state (in other words, the party using the reference state would not be listed as a participant on it). Therefore, it is likely that a transaction containing reference states with two different notaries cannot be committed to the ledger.

As such, if reference states assigned to multiple different notaries are added to a transaction builder then the check below will fail.

**Warning:** Currently, encumbrances should not be used with reference states. In the case where a state is encumbered by an encumbrance state, the encumbrance state should also be referenced in the same transaction that references the encumbered state. This is because the data contained within the encumbered state may take on a different meaning, and likely would do, once the encumbrance state is taken into account.

## 4.1.7 State Pointers

A `StatePointer` contains a pointer to a `ContractState`. The `StatePointer` can be included in a `ContractState` as a property, or included in an off-ledger data structure. `StatePointer`s can be resolved to a `StateAndRef` by performing a look-up. There are two types of pointers; linear and static.

1. `StaticPointer`s are for use with any type of `ContractState`. The `StaticPointer` does as it suggests, it always points to the same `ContractState`.
2. The `LinearPointer` is for use with `LinearStates`. They are particularly useful because due to the way `LinearStates` work, the pointer will automatically point you to the latest version of a `LinearState` that the node performing `resolve` is aware of. In effect, the pointer “moves” as the `LinearState` is updated.

State pointers use `Reference States` to enable the functionality described above. They can be conceptualized as a mechanism to formalise a development pattern where one needs to refer to a specific state from another transaction (`StaticPointer`) or a particular lineage of states (`LinearPointer`). In other words, `StatePointers` do not enable

a feature in Corda which was previously unavailable. Rather, they help to formalise a pattern which was already possible. In that light, it is worth noting some issues which you may encounter in its application:

- If the node calling `resolve` has not seen any transactions containing a `ContractState` which the `StatePointer` points to, then `resolve` will throw an exception. Here, the node calling `resolve` might be missing some crucial data.
- The node calling `resolve` for a `LinearPointer` may have seen and stored transactions containing a `LinearState` with the specified `linearId`. However, there is no guarantee the `StateAndRef<T>` returned by `resolve` is the most recent version of the `LinearState`. The node only returns the most recent version that `_it_` is aware of.

### Resolving state pointers in `TransactionBuilder`

When building transactions, any `StatePointer`s contained within inputs or outputs added to a `TransactionBuilder` can be optionally resolved to reference states using the `resolveStatePointers` method. The effect is that the pointed to data is carried along with the transaction. This may or may not be appropriate in all circumstances, which is why calling the method is optional.

## 4.2 API: Persistence

### Contents

- *API: Persistence*
  - *Schemas*
  - *Custom schema registration*
  - *Object relational mapping*
  - *Persisting Hierarchical Data*
  - *Identity mapping*
  - *JDBC session*
  - *JPA Support*

Corda offers developers the option to expose all or some parts of a contract state to an *Object Relational Mapping* (ORM) tool to be persisted in a *Relational Database Management System* (RDBMS).

The purpose of this, is to assist `Vault` development and allow for the persistence of state data to a custom database table. Persisted states held in the vault are indexed for the purposes of executing queries. This also allows for relational joins between Corda tables and the organization's existing data.

The Object Relational Mapping is specified using [Java Persistence API](#) (JPA) annotations. This mapping is persisted to the database as a table row (a single, implicitly structured data item) by the node automatically every time a state is recorded in the node's local vault as part of a transaction.

---

**Note:** By default, nodes use an H2 database which is accessed using *Java Database Connectivity* JDBC. Any database with a JDBC driver is a candidate and several integrations have been contributed to by the community. Please see the info in “[Node database](#)” for details.

---

## 4.2.1 Schemas

Every `ContractState` may implement the `QueryableState` interface if it wishes to be inserted into a custom table in the node's database and made accessible using SQL.

```
/**
 * A contract state that may be mapped to database schemas configured for this node,
 * to support querying for,
 * or filtering of, states.
 */
@KeepForDJVM
interface QueryableState : ContractState {
    /**
     * Enumerate the schemas this state can export representations of itself as.
     */
    fun supportedSchemas(): Iterable<MappedSchema>

    /**
     * Export a representation for the given schema.
     */
    fun generateMappedObject(schema: MappedSchema): PersistentState
}
```

The `QueryableState` interface requires the state to enumerate the different relational schemas it supports, for instance in situations where the schema has evolved. Each relational schema is represented as a `MappedSchema` object returned by the state's `supportedSchemas` method.

Nodes have an internal `SchemaService` which decides what data to persist by selecting the `MappedSchema` to use. Once a `MappedSchema` is selected, the `SchemaService` will delegate to the `QueryableState` to generate a corresponding representation (mapped object) via the `generateMappedObject` method, the output of which is then passed to the *ORM*.

```
/**
 * A configuration and customisation point for Object Relational Mapping of contract
 * state objects.
 */
interface SchemaService {
    /**
     * Represents any options configured on the node for a schema.
     */
    data class SchemaOptions(val databaseSchema: String? = null, val tablePrefix: String? = null)

    /**
     * Options configured for this node's schemas. A missing entry for a schema
     * implies all properties are null.
     */
    val schemaOptions: Map<MappedSchema, SchemaOptions>

    /**
     * Given a state, select schemas to map it to that are supported by
     * [generateMappedObject] and that are configured
     * for this node.
     */
    fun selectSchemas(state: ContractState): Iterable<MappedSchema>

    /**
     * Map a state to a [PersistentState] for the given schema, either via direct
     * support from the state

```

(continues on next page)

(continued from previous page)

```

    * or via custom logic in this service.
    */
    fun generateMappedObject(state: ContractState, schema: MappedSchema):_>
    ↪PersistentState
}

```

```

/**
 * A database schema that might be configured for this node. As well as a name and
 ↪version for identifying the schema,
 * also list the classes that may be used in the generated object graph in order to
 ↪configure the ORM tool.
 *
 * @param schemaFamily A class to fully qualify the name of a schema family (i.e._
 ↪excludes version)
 * @param version The version number of this instance within the family.
 * @param mappedTypes The JPA entity classes that the ORM layer needs to be configure_
 ↪with for this schema.
 */
@KeepForDJVM
open class MappedSchema(schemaFamily: Class<*>,
    val version: Int,
    val mappedTypes: Iterable<Class<*>>) {
    val name: String = schemaFamily.name

    /**
     * Optional classpath resource containing the database changes for the
     ↪[mappedTypes]
     */
    open val migrationResource: String? = null

    override fun toString(): String = "${this.javaClass.simpleName} (name=$name,_
    ↪version=$version)"

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as MappedSchema

        if (version != other.version) return false
        if (mappedTypes != other.mappedTypes) return false
        if (name != other.name) return false

        return true
    }

    override fun hashCode(): Int {
        var result = version
        result = 31 * result + mappedTypes.hashCode()
        result = 31 * result + name.hashCode()
        return result
    }
}

```

With this framework, the relational view of ledger states can evolve in a controlled fashion in lock-step with internal systems or other integration points and is not dependant on changes to the contract code.

It is expected that multiple contract state implementations might provide mappings within a single schema. For example an Interest Rate Swap contract and an Equity OTC Option contract might both provide a mapping to a Derivative contract within the same schema. The schemas should typically not be part of the contract itself and should exist independently to encourage re-use of a common set within a particular business area or Cordapp.

---

**Note:** It's advisable to avoid cross-references between different schemas as this may cause issues when evolving MappedSchema or migrating its data. At startup, nodes log such violations as warnings stating that there's a cross-reference between MappedSchema's. The detailed messages incorporate information about what schemas, entities and fields are involved.

---

MappedSchema offer a family name that is disambiguated using Java package style name-spacing derived from the class name of a *schema family* class that is constant across versions, allowing the SchemaService to select a preferred version of a schema.

The SchemaService is also responsible for the SchemaOptions that can be configured for a particular MappedSchema. These allow the configuration of database schemas or table name prefixes to avoid clashes with other MappedSchema.

---

**Note:** It is intended that there should be plugin support for the SchemaService to offer version upgrading, implementation of additional schemas, and enable active schemas as being configurable. The present implementation does not include these features and simply results in all versions of all schemas supported by a QueryableState being persisted. This will change in due course. Similarly, the service does not currently support configuring SchemaOptions but will do so in the future.

---

## 4.2.2 Custom schema registration

Custom contract schemas are automatically registered at startup time for CorDapps. The node bootstrap process will scan for states that implement the Queryable state interface. Tables are then created as specified by the MappedSchema identified by each state's supportedSchemas method.

For testing purposes it is necessary to manually register the packages containing custom schemas as follows:

- Tests using MockNetwork and MockNode must explicitly register packages using the *cordappPackages* parameter of MockNetwork
- Tests using MockServices must explicitly register packages using the *cordappPackages* parameter of the MockServices *makeTestDatabaseAndMockServices()* helper method.

---

**Note:** Tests using the DriverDSL will automatically register your custom schemas if they are in the same project structure as the driver call.

---

## 4.2.3 Object relational mapping

To facilitate the ORM, the persisted representation of a QueryableState should be an instance of a PersistentState subclass, constructed either by the state itself or a plugin to the SchemaService. This allows the ORM layer to always associate a StateRef with a persisted representation of a ContractState and allows joining with the set of unconsumed states in the vault.

The PersistentState subclass should be marked up as a JPA 2.1 *Entity* with a defined table name and having properties (in Kotlin, getters/setters in Java) annotated to map to the appropriate columns and SQL types. Additional entities can be included to model these properties where they are more complex, for example collections (Persisting

Hierarchical Data), so the mapping does not have to be *flat*. The `MappedSchema` constructor accepts a list of all JPA entity classes for that schema in the `MappedTypes` parameter. It must provide this list in order to initialise the ORM layer.

Several examples of entities and mappings are provided in the codebase, including `Cash.State` and `CommercialPaper.State`. For example, here's the first version of the cash schema.

```
package net.corda.finance.schemas

import net.corda.core.identity.AbstractParty
import net.corda.core.schemas.MappedSchema
import net.corda.core.schemas.PersistentState
import net.corda.core.serialization.CordaSerializable
import net.corda.core.utilities.MAX_HASH_HEX_SIZE
import net.corda.core.contracts.MAX_ISSUER_REF_SIZE
import org.hibernate.annotations.Type
import javax.persistence.*

/**
 * An object used to fully qualify the [CashSchema] family name (i.e. independent of
 * version).
 */
object CashSchema

/**
 * First version of a cash contract ORM schema that maps all fields of the [Cash]
 * contract state as it stood
 * at the time of writing.
 */
@CordaSerializable
object CashSchemaV1 : MappedSchema(schemaFamily = CashSchema.javaClass, version = 1,
    mappedTypes = listOf(PersistentCashState::class.java)) {

    override val migrationResource = "cash.changelog-master"

    @Entity
    @Table(name = "contract_cash_states", indexes = [Index(name = "ccy_code_idx",
        columnList = "ccy_code"), Index(name = "pennies_idx", columnList = "pennies")])
    class PersistentCashState(
        /** X500Name of owner party */
        @Column(name = "owner_name", nullable = true)
        var owner: AbstractParty?,

        @Column(name = "pennies", nullable = false)
        var pennies: Long,

        @Column(name = "ccy_code", length = 3, nullable = false)
        var currency: String,

        @Column(name = "issuer_key_hash", length = MAX_HASH_HEX_SIZE, nullable =
        false)
        var issuerPartyHash: String,

        @Column(name = "issuer_ref", length = MAX_ISSUER_REF_SIZE, nullable =
        false)
        @Type(type = "corda-wrapper-binary")
        var issuerRef: ByteArray
    ) : PersistentState()
```

(continues on next page)

(continued from previous page)

}

**Note:** If Cordapp needs to be portable between Corda OS (running against H2) and Corda Enterprise (running against a standalone database), consider database vendors specific requirements. Ensure that table and column names are compatible with the naming convention of the database vendors for which the Cordapp will be deployed, e.g. for Oracle database, prior to version 12.2 the maximum length of table/column name is 30 bytes (the exact number of characters depends on the database encoding).

#### 4.2.4 Persisting Hierarchical Data

You may wish to persist hierarchical relationships within states using multiple database tables

You may wish to persist hierarchical relationships within state data using multiple database tables. In order to facilitate this, multiple `PersistentState` subclasses may be implemented. The relationship between these classes is defined using JPA annotations. It is important to note that the `MappedSchema` constructor requires a list of *all* of these subclasses.

An example Schema implementing hierarchical relationships with JPA annotations has been implemented below. This Schema will cause `parent_data` and `child_data` tables to be created.

```
@CordaSerializable
public class SchemaV1 extends MappedSchema {

    /**
     * This class must extend the MappedSchema class. Its name is based on the
     * SchemaFamily name and the associated version number abbreviation (V1, V2... Vn).
     * In the constructor, use the super keyword to call the constructor of
     * MappedSchema with the following arguments: a class literal representing the schema
     * family,
     * a version number and a collection of mappedTypes (class literals) which
     * represent JPA entity classes that the ORM layer needs to be configured with for
     * this schema.
     */

    public SchemaV1() {
        super(Schema.class, 1, ImmutableList.of(PersistentParentToken.class,
        PersistentChildToken.class));
    }

    /**
     * The @entity annotation signifies that the specified POJO class' non-transient
     * fields should be persisted to a relational database using the services
     * of an entity manager. The @table annotation specifies properties of the table
     * that will be created to contain the persisted data, in this case we have
     * specified a name argument which will be used the table's title.
     */

    @Entity
    @Table(name = "parent_data")
    public static class PersistentParentToken extends PersistentState {

        /**
         * The @Column annotations specify the columns that will comprise the
         * inserted table and specify the shape of the fields and associated
         */
    }
}
```

(continues on next page)

(continued from previous page)

```

* data types of each database entry.
 */

@Column(name = "owner") private final String owner;
@Column(name = "issuer") private final String issuer;
@Column(name = "amount") private final int amount;
@Column(name = "linear_id") public final UUID linearId;

/**
 * The @OneToMany annotation specifies a one-to-many relationship between
 →this class and a collection included as a field.
 * The @JoinColumn and @JoinColumns annotations specify on which columns
 →these tables will be joined on.
 */

@OneToMany(cascade = CascadeType.PERSIST)
@JoinColumns({
    @JoinColumn(name = "output_index", referencedColumnName = "output_
→index"),
    @JoinColumn(name = "transaction_id", referencedColumnName =
→"transaction_id"),
})
private final List<PersistentChildToken> listOfPersistentChildTokens;

public PersistentParentToken(String owner, String issuer, int amount, UUID_
→linearId, List<PersistentChildToken> listOfPersistentChildTokens) {
    this.owner = owner;
    this.issuer = issuer;
    this.amount = amount;
    this.linearId = linearId;
    this.listOfPersistentChildTokens = listOfPersistentChildTokens;
}

// Default constructor required by hibernate.
public PersistentParentToken() {
    this.owner = "";
    this.issuer = "";
    this.amount = 0;
    this.linearId = UUID.randomUUID();
    this.listOfPersistentChildTokens = null;
}

public String getOwner() {
    return owner;
}

public String getIssuer() {
    return issuer;
}

public int getAmount() {
    return amount;
}

public UUID getLinearId() {
    return linearId;
}

```

(continues on next page)

(continued from previous page)

```

public List<PersistentChildToken> getChildTokens() { return
    ↪listOfPersistentChildTokens; }

}

@Entity
@CordaSerializable
@Table(name = "child_data")
public static class PersistentChildToken {
    // The @Id annotation marks this field as the primary key of the persisted
    ↪entity.
    @Id
    private final UUID Id;
    @Column(name = "owner")
    private final String owner;
    @Column(name = "issuer")
    private final String issuer;
    @Column(name = "amount")
    private final int amount;

    /**
     * The @ManyToOne annotation specifies that this class will be present as a
     ↪member of a collection on a parent class and that it should
     * be persisted with the joining columns specified in the parent class. It is
     ↪important to note the targetEntity parameter which should correspond
     * to a class literal of the parent class.
     */
}

@ManyToOne(targetEntity = PersistentParentToken.class)
private final TokenState persistentParentToken;

public PersistentChildToken(String owner, String issuer, int amount) {
    this.Id = UUID.randomUUID();
    this.owner = owner;
    this.issuer = issuer;
    this.amount = amount;
    this.persistentParentToken = null;
}

// Default constructor required by hibernate.
public PersistentChildToken() {
    this.Id = UUID.randomUUID();
    this.owner = "";
    this.issuer = "";
    this.amount = 0;
    this.persistentParentToken = null;
}

public UUID getId() {
    return Id;
}

public String getOwner() {
    return owner;
}

```

(continues on next page)

(continued from previous page)

```

public String getIssuer() {
    return issuer;
}

public int getAmount() {
    return amount;
}

public TokenState getPersistentToken() {
    return persistentToken;
}

}

}

```

```

@CordaSerializable
object SchemaV1 : MappedSchema(schemaFamily = Schema::class.java, version = 1,
    ↪mappedTypes = listOf(PersistentParentToken::class.java, PersistentChildToken::class.
    ↪java))

@Entity
@Table(name = "parent_data")
class PersistentParentToken(
    @Column(name = "owner")
    var owner: String,

    @Column(name = "issuer")
    var issuer: String,

    @Column(name = "amount")
    var currency: Int,

    @Column(name = "linear_id")
    var linear_id: UUID,

    @JoinColumns(JoinColumn(name = "transaction_id", referencedColumnName =
    ↪"transaction_id"), JoinColumn(name = "output_index", referencedColumnName = "output_
    ↪index"))

    var listOfPersistentChildTokens: MutableList<PersistentChildToken>
) : PersistentState()

@Entity
@CordaSerializable
@Table(name = "child_data")
class PersistentChildToken(
    @Id
    var Id: UUID = UUID.randomUUID(),

    @Column(name = "owner")
    var owner: String,

    @Column(name = "issuer")
    var issuer: String,

```

(continues on next page)

(continued from previous page)

```

    @Column(name = "amount")
    var currency: Int,

    @Column(name = "linear_id")
    var linear_id: UUID,

    @ManyToOne(targetEntity = PersistentParentToken::class)
    var persistentParentToken: TokenState

) : PersistentState()

```

## 4.2.5 Identity mapping

Schema entity attributes defined by identity types (`AbstractParty`, `Party`, `AnonymousParty`) are automatically processed to ensure only the `X500Name` of the identity is persisted where an identity is well known, otherwise a null value is stored in the associated column. To preserve privacy, identity keys are never persisted. Developers should use the `IdentityService` to resolve keys from well known X500 identity names.

## 4.2.6 JDBC session

Apps may also interact directly with the underlying Node's database by using a standard JDBC connection (session) as described by the [Java SQL Connection API](#)

Use the `ServiceHub.jdbcSession` function to obtain a JDBC connection as illustrated in the following example:

```

    val nativeQuery = "SELECT v.transaction_id, v.output_index FROM vault_states_
    ↪v WHERE v.state_status = 0"

    database.transaction {
        val jdbcSession = services.jdbcSession()
        val prepStatement = jdbcSession.prepareStatement(nativeQuery)
        val rs = prepStatement.executeQuery()
    }

```

JDBC sessions can be used in flows and services (see “[Writing flows](#)”).

The following example illustrates the creation of a custom Corda service using a `jdbcSession`:

```

object CustomVaultQuery {

    @CordaService
    class Service(val services: AppServiceHub) : SingletonSerializeAsToken() {
        private companion object {
            private val log = contextLogger()
        }

        fun rebalanceCurrencyReserves(): List<Amount<Currency>> {
            val nativeQuery = """
                select
                    cashschema.ccy_code,
                    sum(cashschema.pennies)
                from
                    vault_states vaultschema
                join
                    contract_cash_states cashschema
            """

```

(continues on next page)

(continued from previous page)

```

        where
            vaultschema.output_index=cashschema.output_index
            and vaultschema.transaction_id=cashschema.transaction_id
            and vaultschema.state_status=0
        group by
            cashschema.ccy_code
        order by
            sum(cashschema.pennies) desc
    """
    log.info("SQL to execute: $nativeQuery")
    val session = services.jdbcSession()
    return session.prepareStatement(nativeQuery).use { prepStatement ->
        prepStatement.executeQuery().use { rs ->
            val topUpLimits: MutableList<Amount<Currency>> = mutableListOf()
            while (rs.next()) {
                val currencyStr = rs.getString(1)
                val amount = rs.getLong(2)
                log.info("$currencyStr : $amount")
                topUpLimits.add(Amount(amount, Currency.
→getInstance(currencyStr)))
            }
            topUpLimits
        }
    }
}

```

which is then referenced within a custom flow:

```
@Suspendable
@Throws(CashException::class)
override fun call(): List<SignedTransaction> {
    progressTracker.currentStep = AWAITING_REQUEST
    val topupRequest = otherPartySession.receive<TopupRequest>().unwrap {
        it
    }
}

    val customVaultQueryService = serviceHub.cordaService(CustomVaultQuery.
→Service::class.java)
    val reserveLimits = customVaultQueryService.rebalanceCurrencyReserves()

    val txns: List<SignedTransaction> = reserveLimits.map { amount ->
        // request asset issue
        logger.info("Requesting currency issue $amount")
        val txn = issueCashTo(amount, topupRequest.issueToParty, topupRequest.
→issuerPartyRef, topupRequest.notaryParty)
        progressTracker.currentStep = SENDING_TOP_UP_ISSUE_REQUEST
        return@map txn.stx
    }

    otherPartySession.send(txns)
    return txns
}
```

For examples on testing @CordaService implementations, see the oracle example [here](#).

## 4.2.7 JPA Support

In addition to `jdbcTemplate`, ServiceHub also exposes the Java Persistence API to flows via the `withEntityManager` method. This method can be used to persist and query entities which inherit from `MappedSchema`. This is particularly useful if off-ledger data must be maintained in conjunction with on-ledger state data.

---

**Note:** Your entity must be included as a `mappedType` as part of a `MappedSchema` for it to be added to Hibernate as a custom schema. If it's not included as a `mappedType`, a corresponding table will not be created. See Samples below.

---

The code snippet below defines a `PersistentFoo` type inside `FooSchemaV1`. Note that `PersistentFoo` is added to a list of mapped types which is passed to `MappedSchema`. This is exactly how state schemas are defined, except that the entity in this case should not subclass `PersistentState` (as it is not a state object). See examples:

```
public class FooSchema {}  
  
public class FooSchemaV1 extends MappedSchema {  
    FooSchemaV1() {  
        super(FooSchema.class, 1, ImmutableList.of(PersistentFoo.class));  
    }  
  
    @Entity  
    @Table(name = "foos")  
    class PersistentFoo implements Serializable {  
        @Id  
        @Column(name = "foo_id")  
        String fooId;  
  
        @Column(name = "foo_data")  
        String fooData;  
    }  
}
```

```
object FooSchema  
  
object FooSchemaV1 : MappedSchema(schemaFamily = FooSchema.javaClass, version = 1,  
    ↪ mappedTypes = listOf(PersistentFoo::class.java)) {  
    @Entity  
    @Table(name = "foos")  
    class PersistentFoo(@Id @Column(name = "foo_id") var fooId: String, @Column(name =  
    ↪ "foo_data") var fooData: String) : Serializable  
}
```

Instances of `PersistentFoo` can be manually persisted inside a flow as follows:

```
PersistentFoo foo = new PersistentFoo(new UniqueIdentifier().getId().toString(), "Bar"  
    ↪ );  
serviceHub.withEntityManager(entityManager -> {  
    entityManager.persist(foo);  
    return null;  
});
```

```
val foo = FooSchemaV1.PersistentFoo(UniqueIdentifier().id.toString(), "Bar")  
serviceHub.withEntityManager {
```

(continues on next page)

(continued from previous page)

```
    persist(foo)  
}
```

And retrieved via a query, as follows:

```
node.getServices().withEntityManager((EntityManager entityManager) -> {  
    CriteriaQuery<PersistentFoo> query = entityManager.getCriteriaBuilder().  
        createQuery(PersistentFoo.class);  
    Root<PersistentFoo> type = query.from(PersistentFoo.class);  
    query.select(type);  
    return entityManager.createQuery(query).getResultList();  
});
```

```
val result: MutableList<FooSchemaV1.PersistentFoo> = services.withEntityManager {  
    val query = criteriaBuilder.createQuery(FooSchemaV1.PersistentFoo::class.java)  
    val type = query.from(FooSchemaV1.PersistentFoo::class.java)  
    query.select(type)  
    createQuery(query).resultList  
}
```

Please note that suspendable flow operations such as:

- FlowSession.send
- FlowSession.receive
- FlowLogic.receiveAll
- FlowLogic.sleep
- FlowLogic.subFlow

Cannot be used within the lambda function passed to withEntityManager.

## 4.3 API: Contracts

---

**Note:** Before reading this page, you should be familiar with the key concepts of *Contracts*.

---

### Contents

- *API: Contracts*
  - *Contract*
  - *LedgerTransaction*
  - *requireThat*
  - *Commands*
    - \* *Branching verify with commands*

### 4.3.1 Contract

Contracts are classes that implement the `Contract` interface. The `Contract` interface is defined as follows:

```
/**
 * Implemented by a program that implements business logic on the shared ledger. All
 * participants run this code for
 * every [net.corda.core.transactions.LedgerTransaction] they see on the network, for
 * every input and output state. All
 * contracts must accept the transaction for it to be accepted: failure of any aborts
 * the entire thing. The time is taken
 * from a trusted time-window attached to the transaction itself i.e. it is NOT
 * necessarily the current time.
 *
 * TODO: Contract serialization is likely to change, so the annotation is likely
 * temporary.
 */
@KeepForDJVM
@CordaSerializable
interface Contract {
    /**
     * Takes an object that represents a state transition, and ensures the inputs/
     * outputs/commands make sense.
     * Must throw an exception if there's a problem that should prevent state
     * transition. Takes a single object
     * rather than an argument so that additional data can be added without breaking
     * binary compatibility with
     * existing contract code.
     */
    @Throws(IllegalArgumentException::class)
    fun verify(tx: LedgerTransaction)
}
```

`Contract` has a single method, `verify`, which takes a `LedgerTransaction` as input and returns nothing. This function is used to check whether a transaction proposal is valid, as follows:

- We gather together the contracts of each of the transaction's input and output states
- We call each contract's `verify` function, passing in the transaction as an input
- The proposal is only valid if none of the `verify` calls throw an exception

`verify` is executed in a sandbox:

- It does not have access to the enclosing scope
- The libraries available to it are whitelisted to disallow:
  - \* Network access
  - \* I/O such as disk or database access
  - \* Sources of randomness such as the current time or random number generators

This means that `verify` only has access to the properties defined on `LedgerTransaction` when deciding whether a transaction is valid.

Here are the two simplest `verify` functions:

- A `verify` that **accepts** all possible transactions:

```
override fun verify(tx: LedgerTransaction) {
    // Always accepts!
}
```

```
@Override
public void verify(LedgerTransaction tx) {
    // Always accepts!
}
```

- A verify that **rejects** all possible transactions:

```
override fun verify(tx: LedgerTransaction) {
    throw IllegalArgumentException("Always rejects!")
}
```

```
@Override
public void verify(LedgerTransaction tx) {
    throw new IllegalArgumentException("Always rejects!");
}
```

### 4.3.2 LedgerTransaction

The LedgerTransaction object passed into verify has the following properties:

```
/** The resolved input states which will be consumed/invalidated by the
 * execution of this transaction. */
override val inputs: List<StateAndRef<ContractState>>,
/** The outputs created by the transaction. */
override val outputs: List<TransactionState<ContractState>>,
/** Arbitrary data passed to the program of each input state. */
val commands: List<CommandWithParties<CommandData>>,
/** A list of [Attachment] objects identified by the transaction that are
 * needed for this transaction to verify. */
val attachments: List<Attachment>,
/** The hash of the original serialised WireTransaction. */
override val id: SecureHash,
/** The notary that the tx uses, this must be the same as the notary of all
 * the inputs, or null if there are no inputs. */
override val notary: Party?,
/** The time window within which the tx is valid, will be checked against
 * notary pool member clocks. */
val timeWindow: TimeWindow?,
/** Random data used to make the transaction hash unpredictable even if the
 * contents can be predicted; needed to avoid some obscure attacks. */
val privacySalt: PrivacySalt,
/**
 * Network parameters that were in force when the transaction was constructed.
 * This is nullable only for backwards
 * compatibility for serialized transactions. In reality this field will
 * always be set when on the normal codepaths.
 */
override val networkParameters: NetworkParameters?,
/** Referenced states, which are like inputs but won't be consumed. */
override val references: List<StateAndRef<ContractState>>
```

Where:

- inputs are the transaction's inputs as List<StateAndRef<ContractState>>
- outputs are the transaction's outputs as List<TransactionState<ContractState>>

- commands are the transaction's commands and associated signers, as `List<CommandWithParties<CommandData>>`
- attachments are the transaction's attachments as `List<Attachment>`
- notary is the transaction's notary. This must match the notary of all the inputs
- timeWindow defines the window during which the transaction can be notarised

`LedgerTransaction` exposes a large number of utility methods to access the transaction's contents:

- `inputStates` extracts the input `ContractState` objects from the list of `StateAndRef`
- `getInput/getOutput/getCommand/getAttachment` extracts a component by index
- `getAttachment` extracts an attachment by ID
- `inputsOfType/inRefsOfType/outputsOfType/outRefsOfType/commandsOfType` extracts components based on their generic type
- `filterInputs/filterInRefs/filterOutputs/filterOutRefs/filterCommands` extracts components based on a predicate
- `findInput/findInRef/findOutput/findOutRef/findCommand` extracts the single component that matches a predicate, or throws an exception if there are multiple matches

### 4.3.3 requireThat

`verify` can be written to manually throw an exception for each constraint:

```
override fun verify(tx: LedgerTransaction) {
    if (tx.inputs.size > 0)
        throw IllegalArgumentException("No inputs should be consumed when issuing an
→X.");
    if (tx.outputs.size != 1)
        throw IllegalArgumentException("Only one output state should be created.")
}
```

```
public void verify(LedgerTransaction tx) {
    if (tx.getInputs().size() > 0)
        throw new IllegalArgumentException("No inputs should be consumed when issuing
→an X.");
    if (tx.getOutputs().size() != 1)
        throw new IllegalArgumentException("Only one output state should be created.
→");
}
```

However, this is verbose. To impose a series of constraints, we can use `requireThat` instead:

```
requireThat {
    "No inputs should be consumed when issuing an X." using (tx.inputs.isEmpty())
    "Only one output state should be created." using (tx.outputs.size == 1)
    val out = tx.outputs.single() as XState
    "The sender and the recipient cannot be the same entity." using (out.sender !=
→out.recipient)
    "All of the participants must be signers." using (command.signers.containsAll(out.
→participants))
}
```

(continues on next page)

(continued from previous page)

```
"The X's value must be non-negative." using (out.x.value > 0)
}
```

```
requireThat(require -> {
    require.using("No inputs should be consumed when issuing an X.", tx.getInputs().  
isEmpty());
    require.using("Only one output state should be created.", tx.getOutputs().size()  
== 1);
    final XState out = (XState) tx.getOutputs().get(0);
    require.using("The sender and the recipient cannot be the same entity.", out.  
getSender() != out.getRecipient());
    require.using("All of the participants must be signers.", command.getSigners().  
containsAll(out.getParticipants()));
    require.using("The X's value must be non-negative.", out.getX().getValue() > 0);
    return null;
});
```

For each `<String, Boolean>` pair within `requireThat`, if the boolean condition is false, an `IllegalArgumentException` is thrown with the corresponding string as the exception message. In turn, this exception will cause the transaction to be rejected.

#### 4.3.4 Commands

`LedgerTransaction` contains the commands as a list of `CommandWithParties` instances. `CommandWithParties` pairs a `CommandData` with a list of required signers for the transaction:

```
/** A [Command] where the signing parties have been looked up if they have a well-  
known/recognised institutional key. */
@KeepForDJVM
@CordaSerializable
data class CommandWithParties<out T : CommandData>(  
    val signers: List<PublicKey>,  
    /** If any public keys were recognised, the looked up institutions are  
     * available here */  
    @Deprecated("Should not be used in contract verification code as it is non-  
    deterministic, will be disabled for some future target platform version onwards and  
    will take effect only for CorDapps targeting those versions.")  
    val signingParties: List<Party>,  
    val value: T
)
```

Where:

- `signers` is the list of each signer's `PublicKey`
- `signingParties` is the list of the signer's identities, if known
- `value` is the object being signed (a command, in this case)

#### Branching verify with commands

Generally, we will want to impose different constraints on a transaction based on its commands. For example, we will want to impose different constraints on a cash issuance transaction to on a cash transfer transaction.

We can achieve this by extracting the command and using standard branching logic within `verify`. Here, we extract the single command of type `XContract.Commands` from the transaction, and branch `verify` accordingly:

```

class XContract : Contract {
    interface Commands : CommandData {
        class Issue : TypeOnlyCommandData(), Commands
        class Transfer : TypeOnlyCommandData(), Commands
    }

    override fun verify(tx: LedgerTransaction) {
        val command = tx.findCommand<Commands> { true }

        when (command.value) {
            is Commands.Issue -> {
                // Issuance verification logic.
            }
            is Commands.Transfer -> {
                // Transfer verification logic.
            }
        }
    }
}

```

```

public class XContract implements Contract {
    public interface Commands extends CommandData {
        class Issue extends TypeOnlyCommandData implements Commands {}
        class Transfer extends TypeOnlyCommandData implements Commands {}
    }

    @Override
    public void verify(LedgerTransaction tx) {
        final Commands command = tx.findCommand(Commands.class, cmd -> true).
        ↪getValue();

        if (command instanceof Commands.Issue) {
            // Issuance verification logic.
        } else if (command instanceof Commands.Transfer) {
            // Transfer verification logic.
        }
    }
}

```

## 4.4 API: Contract Constraints

**Note:** Before reading this page, you should be familiar with the key concepts of [Contracts](#).

### Contents

- *API: Contract Constraints*
  - *Reasons for Contract Constraints*
    - \* *Implicit vs Explicit Contract upgrades*
  - *Types of Contract Constraints*

- *Signature Constraints*
  - \* *Signing CorDapps for use with Signature Constraints*
  - \* *Using Signature Constraints in transactions*
  - \* *App versioning with Signature Constraints*
- *Hash Constraints*
  - \* *Issues when using the HashAttachmentConstraint*
  - \* *Hash constrained states in private networks*
- *Contract/State Agreement*
- *Using Contract Constraints in Transactions*
- *CorDapps as attachments*
- *Constraints propagation*
- *Constraints migration to Corda 4*
- *Debugging*

### 4.4.1 Reasons for Contract Constraints

*Contract constraints* solve two problems faced by any decentralised ledger that supports evolution of data and code:

1. Controlling and agreeing upon upgrades
2. Preventing attacks

Upgrades and security are intimately related because if an attacker can “upgrade” your data to a version of an app that gives them a back door, they would be able to do things like print money or edit states in any way they want. That’s why it’s important for participants of a state to agree on what kind of upgrades will be allowed.

Every state on the ledger contains the fully qualified class name of a *Contract* implementation, and also a *constraint*. This constraint specifies which versions of an application can be used to provide the named class, when the transaction is built. New versions released after a transaction is signed and finalised won’t affect prior transactions because the old code is attached to it.

#### Implicit vs Explicit Contract upgrades

Constraints are not the only way to manage upgrades to transactions. There are two ways of handling upgrades to a smart contract in Corda:

- **Implicit:** By pre-authorising multiple implementations of the contract ahead of time, using constraints.
- **Explicit:** By creating a special *contract upgrade transaction* and getting all participants of a state to sign it using the contract upgrade flows.

The advantage of pre-authorising upgrades using constraints is that you don’t need the heavyweight process of creating upgrade transactions for every state on the ledger. The disadvantage is that you place more faith in third parties, who could potentially change the app in ways you did not expect or agree with. The advantage of using the explicit upgrade approach is that you can upgrade states regardless of their constraint, including in cases where you didn’t anticipate a need to do so. But it requires everyone to sign, manually authorise the upgrade, consumes notary and ledger resources, and is just in general more complex.

This article focuses on the first approach. To learn about the second please see [Release new CorDapp versions](#).

## 4.4.2 Types of Contract Constraints

Corda supports several types of constraints to cover a wide set of client requirements:

- **Hash constraint:** Exactly one version of the app can be used with this state. This prevents the app from being upgraded in the future while still making use of the state created with the original version.
- **Compatibility zone whitelisted (or CZ whitelisted) constraint:** The compatibility zone operator lists the hashes of the versions that can be used with a contract class name.
- **Signature constraint:** Any version of the app signed by the given CompositeKey can be used. This allows app issuers to express the complex social and business relationships that arise around code ownership. For example, a Signature Constraint allows a new version of an app to be produced and applied to an existing state as long as it has been signed by the same key(s) as the original version.
- **Always accept constraint:** Any version of the app can be used. This is insecure but convenient for testing.

## 4.4.3 Signature Constraints

The best kind of constraint to use is the **Signature Constraint**. If you sign your application it will be used automatically. We recommend signature constraints because they let you express complex social and business relationships while allowing smooth migration of existing data to new versions of your application.

Signature constraints can specify flexible threshold policies, but if you use the automatic support then a state will require the attached app to be signed by every key that the first attachment was signed by. Thus if the app that was used to issue the states was signed by Alice and Bob, every transaction must use an attachment signed by Alice and Bob. Doing so allows the app to be upgraded and changed while still remaining valid for use with the previously issued states.

More complex policies can be expressed through Signature Constraints if required. Allowing policies where only a number of the possible signers must sign the new version of an app that is interacting with previously issued states. Accepting different versions of apps in this way makes it possible for multiple versions to be valid across the network as long as the majority (or possibly a minority) agree with the logic provided by the apps.

Hash and zone whitelist constraints are left over from earlier Corda versions before Signature Constraints were implemented. They make it harder to upgrade applications than when using signature constraints, so they're best avoided.

Further information into the design of Signature Constraints can be found in its design document.

### **S**igning CorDapps for use with Signature Constraints

Expanding on the previous section, for an app to use Signature Constraints, it must be signed by a CompositeKey or a simpler PublicKey. The signers of the app can consist of a single organisation or multiple organisations. Once the app has been signed, it can be distributed across the nodes that intend to use it.

Each transaction received by a node will then verify that the apps attached to it have the correct signers as specified by its Signature Constraints. This ensures that the version of each app is acceptable to the transaction's input states.

More information on how to sign an app directly from Gradle can be found in the [CorDapp Jar signing](#) section of the documentation.

### **U**sing Signature Constraints in transactions

If the app is signed, Signature Constraints will be used by default (in most situations) by the TransactionBuilder when adding output states. This is expanded upon in [Using Contract Constraints in Transactions](#).

---

**Note:** Signature Constraints are used by default except when a new transaction contains an input state with a Hash Constraint. In this situation the Hash Constraint is used.

---

### App versioning with Signature Constraints

Signed apps require a version number to be provided, see [Versioning](#). You can't import two different JARs that claim to be the same version, provide the same contract classes and which are both signed. At runtime the node will throw a `DuplicateContractClassException` exception if this condition is violated.

## 4.4.4 Hash Constraints

### Issues when using the HashAttachmentConstraint

When setting up a new network, it is possible to encounter errors when states are issued with the `HashAttachmentConstraint`, but not all nodes have that same version of the CorDapp installed locally.

In this case, flows will fail with a `ContractConstraintRejection`, and are sent to the flow hospital. From there, they are suspended, waiting to be retried on node restart. This gives the node operator the opportunity to recover from those errors, which in the case of constraint violations means adding the right cordapp jar to the `cordapps` folder.

### Hash constrained states in private networks

Where private networks started life using CorDapps with hash constrained states, we have introduced a mechanism to relax the checking of these hash constrained states when upgrading to signed CorDapps using signature constraints.

The Java system property `-Dnet.corda.node.disableHashConstraints="true"` may be set to relax the hash constraint checking behaviour. For this to work, every participant of the network must set the property to the same value. Therefore, this mode should only be used upon “out of band” agreement by all participants in a network.

**Warning:** This flag should remain enabled until every hash constrained state is exited from the ledger.

## 4.4.5 Contract/State Agreement

Starting with Corda 4, a `ContractState` must explicitly indicate which `Contract` it belongs to. When a transaction is verified, the contract bundled with each state in the transaction must be its “owning” contract, otherwise we cannot guarantee that the transition of the `ContractState` will be verified against the business rules that should apply to it.

There are two mechanisms for indicating ownership. One is to annotate the `ContractState` with the `BelongsToContract` annotation, indicating the `Contract` class to which it is tied:

```
@BelongsToContract (MyContract.class)
public class MyState implements ContractState {
    // implementation goes here
}
```

```
@BelongsToContract(MyContract::class)
data class MyState(val value: Int) : ContractState {
    // implementation goes here
}
```

The other is to define the ContractState class as an inner class of the Contract class

```
public class MyContract implements Contract {

    public static class MyState implements ContractState {
        // state implementation goes here
    }

    // contract implementation goes here
}
```

```
class MyContract : Contract {
    data class MyState(val value: Int) : ContractState
}
```

If a ContractState's owning Contract cannot be identified by either of these mechanisms, and the targetVersion of the CorDapp is 4 or greater, then transaction verification will fail with a TransactionRequiredContractUnspecifiedException. If the owning Contract *can* be identified, but the ContractState has been bundled with a different contract, then transaction verification will fail with a TransactionContractConflictException.

#### 4.4.6 Using Contract Constraints in Transactions

The app version used by a transaction is defined by its attachments. The JAR containing the state and contract classes, and optionally its dependencies, are all attached to the transaction. Nodes will download this JAR from other nodes if they haven't seen it before, so it can be used for verification.

The TransactionBuilder will manage the details of constraints for you, by selecting both constraints and attachments to ensure they line up correctly. Therefore you only need to have a basic understanding of this topic unless you are doing something sophisticated.

By default the TransactionBuilder will use *Signature Constraints* for any issuance transactions if the app attached to it is signed.

To manually define the Contract Constraint of an output state, see the example below:

```
TransactionBuilder transaction() {
    TransactionBuilder transaction = new TransactionBuilder(notary());
    // Signature Constraint used if app is signed
    transaction.addOutputState(state);
    // Explicitly using a Signature Constraint
    transaction.addOutputState(state, CONTRACT_ID, new_
    ↪SignatureAttachmentConstraint(getOurIdentity().getOwningKey()));
    // Explicitly using a Hash Constraint
    transaction.addOutputState(state, CONTRACT_ID, new_
    ↪HashAttachmentConstraint getServiceHub().getCordappProvider().
    ↪getContractAttachmentID(CONTRACT_ID));
    // Explicitly using a Whitelisted by Zone Constraint
    transaction.addOutputState(state, CONTRACT_ID,_
    ↪WhitelistedByZoneAttachmentConstraint.INSTANCE);
    // Explicitly using an Always Accept Constraint
```

(continues on next page)

(continued from previous page)

```

    transaction.addOutputState(state, CONTRACT_ID, AlwaysAcceptAttachmentConstraint.
→INSTANCE);

    // other transaction stuff
    return transaction;
}

```

```

private fun transaction(): TransactionBuilder {
    val transaction = TransactionBuilder(notary())
    // Signature Constraint used if app is signed
    transaction.addOutputState(state)
    // Explicitly using a Signature Constraint
    transaction.addOutputState(state, constraint =
→SignatureAttachmentConstraint(ourIdentity.owningKey))
    // Explicitly using a Hash Constraint
    transaction.addOutputState(state, constraint =
→HashAttachmentConstraint(serviceHub.cordappProvider.
→getContractAttachmentID(CONTRACT_ID) !!))
    // Explicitly using a Whitelisted by Zone Constraint
    transaction.addOutputState(state, constraint =
→WhitelistedByZoneAttachmentConstraint)
    // Explicitly using an Always Accept Constraint
    transaction.addOutputState(state, constraint = AlwaysAcceptAttachmentConstraint)

    // other transaction stuff
    return transaction
}

```

#### 4.4.7 CorDapps as attachments

CorDapp JARs (see [What is a CorDapp?](#)) that contain classes implementing the `Contract` interface are automatically loaded into the `AttachmentStorage` of a node, and made available as `ContractAttachments`.

They are retrievable by hash using `AttachmentStorage.openAttachment`. These JARs can either be installed on the node or will be automatically fetched over the network when receiving a transaction.

**Warning:** The obvious way to write a CorDapp is to put all your states, contracts, flows and support code into a single Java module. This will work but it will effectively publish your entire app onto the ledger. That has two problems: (1) it is inefficient, and (2) it means changes to your flows or other parts of the app will be seen by the ledger as a “new app”, which may end up requiring essentially unnecessary upgrade procedures. It’s better to split your app into multiple modules: one which contains just states, contracts and core data types. And another which contains the rest of the app. See [Modules](#).

#### 4.4.8 Constraints propagation

As was mentioned above, the `TransactionBuilder` API gives the CorDapp developer or even malicious node owner the possibility to construct output states with a constraint of their choosing.

For the ledger to remain in a consistent state, the expected behavior is for output state to inherit the constraints of input states. This guarantees that for example, a transaction can’t output a state with the `AlwaysAcceptAttachmentConstraint` when the corresponding input state was the

`SignatureAttachmentConstraint`. Translated, this means that if this rule is enforced, it ensures that the output state will be spent under similar conditions as it was created.

Before version 4, the constraint propagation logic was expected to be enforced in the contract verify code, as it has access to the entire Transaction.

Starting with version 4 of Corda the constraint propagation logic has been implemented and enforced directly by the platform, unless disabled by putting `@NoConstraintPropagation` on the Contract class which reverts to the previous behavior of expecting apps to do this.

For contracts that are not annotated with `@NoConstraintPropagation`, the platform implements a fairly simple constraint transition policy to ensure security and also allow the possibility to transition to the new `SignatureAttachmentConstraint`.

During transaction building the `AutomaticPlaceholderConstraint` for output states will be resolved and the best contract attachment versions will be selected based on a variety of factors so that the above holds true. If it can't find attachments in storage or there are no possible constraints, the `TransactionBuilder` will throw an exception.

#### 4.4.9 Constraints migration to Corda 4

Please read [CorDapp constraints migration](#) to understand how to consume and evolve pre-Corda 4 issued hash or CZ whitelisted constrained states using a Corda 4 signed CorDapp (using signature constraints).

#### 4.4.10 Debugging

If an attachment constraint cannot be resolved, a `MissingContractAttachments` exception is thrown. There are three common sources of `MissingContractAttachments` exceptions:

You are running a test and have not specified the CorDapp packages to scan. When using `MockNetwork` ensure you have provided a package containing the contract class in `MockNetworkParameters`. See [API: Testing](#).

Similarly package names need to be provided when testing using `DriverDSL`. `DriverParameters` has a property `cordappsForAllNodes` (Kotlin) or method `withCordappsForAllNodes` in Java. Pass the collection of `TestCordapp` created by utility method `TestCordapp.findCordapp(String)`.

Example of creation of two Cordapps with Finance App Flows and Finance App Contracts in Kotlin:

```
Driver.driver(DriverParameters(cordappsForAllNodes = listOf(TestCordapp.
    ↪findCordapp("net.corda.finance.schemas"),
    TestCordapp.findCordapp("net.corda.finance.flows")))) {
    // Your test code goes here
}
```

The same example in Java:

```
Driver.driver(new DriverParameters()
    .withCordappsForAllNodes(Arrays.asList(TestCordapp.findCordapp("net.
    ↪corda.finance.schemas"),
    TestCordapp.findCordapp("net.corda.finance.flows"))), dsl -> {
    // Your test code goes here
});
```

When running the Corda node ensure all CordDapp JARs are placed in `cordapps` directory of each node. By default Gradle Cordform task `deployNodes` copies all JARs if CorDapps to deploy are specified. See [Creating nodes locally](#) for detailed instructions.

You are specifying the fully-qualified name of the contract incorrectly. For example, you've defined `MyContract` in the package `com.mycompany.myapp.contracts`, but the fully-qualified contract name you pass to the

TransactionBuilder is com.mycompany.myapp.MyContract (instead of com.mycompany.myapp.contracts.MyContract).

## 4.5 API: Vault Query

### Contents

- *API: Vault Query*
  - *Overview*
  - *Pagination*
  - *Example usage*
    - \* *Kotlin*
    - \* *Java examples*
  - *Troubleshooting*
  - *Behavioural notes*
  - *Other use case scenarios*
  - *Mapping owning keys to external IDs*

### 4.5.1 Overview

Corda has been architected from the ground up to encourage usage of industry standard, proven query frameworks and libraries for accessing RDBMS backed transactional stores (including the Vault).

Corda provides a number of flexible query mechanisms for accessing the Vault:

- Vault Query API
- Using a JDBC session (as described in *Persistence*)
- Custom JPA/JPQL queries
- Custom 3rd party Data Access frameworks such as [Spring Data](#)

The majority of query requirements can be satisfied by using the Vault Query API, which is exposed via the `VaultService` for use directly by flows:

```
/**  
 * 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: [StateStatus.UNCONSUMED], [StateStatus.  
 * CONSUMED], [StateStatus.ALL].
```

(continues on next page)

(continued from previous page)

```

* 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].
*/
@Throws(VaultQueryException::class)
fun <T : ContractState> _queryBy(criteria: QueryCriteria,
                                         paging: PageSpecification,
                                         sorting: Sort,
                                         contractStateType: Class<out T>): Vault.Page<T>

/**
 * 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 [queryBy]).
* 2) an [Observable] of [Vault.Update].
*
* @throws VaultQueryException if the query cannot be executed for any reason.
*
* Notes:
* - The snapshot part of the query adheres to the same behaviour as the [queryBy]_
* function.
* - The update part of the query currently only supports query criteria filtering_
* by contract
* type(s) and state status(es). CID-731 <https://r3-cev.atlassian.net/browse/CID-731> proposes
* adding the complete set of [QueryCriteria] filtering.
*/
@Throws(VaultQueryException::class)
fun <T : ContractState> _trackBy(criteria: QueryCriteria,
                                         paging: PageSpecification,
                                         sorting: Sort,
                                         contractStateType: Class<out T>): DataFeed<Vault.Page
                                         <T>, Vault.Update<T>>

```

And via CordaRPCOps for use by RPC client applications:

```

@RPCReturnsObservables
fun <T : ContractState> vaultQueryBy(criteria: QueryCriteria,
                                         paging: PageSpecification,
                                         sorting: Sort,
                                         contractStateType: Class<out T>): Vault.Page<T>

```

```
@RPCReturnsObservables
fun <T : ContractState> vaultTrackBy(criteria: QueryCriteria,
                                         paging: PageSpecification,
                                         sorting: Sort,
                                         contractStateType: Class<out T>): DataFeed<Vault.
                                         ↪Page<T>, Vault.Update<T>>
```

Helper methods are also provided with default values for arguments:

```
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> vaultQueryWithPagingSpec(contractStateType: Class<out T>, ↪
                                         ↪criteria: QueryCriteria, paging: PageSpecification): Vault.Page<T>

fun <T : ContractState> vaultQueryWithSorting(contractStateType: Class<out T>, ↪
                                         ↪criteria: QueryCriteria, sorting: Sort): Vault.Page<T>
```

```
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>>
```

The API provides both static (snapshot) and dynamic (snapshot with streaming updates) methods for a defined set of filter criteria:

- Use `queryBy` to obtain a current snapshot of data (for a given `QueryCriteria`)
- Use `trackBy` to obtain both a current snapshot and a future stream of updates (for a given `QueryCriteria`)

---

**Note:** Streaming updates are only filtered based on contract type and state status (UNCONSUMED, CONSUMED, ALL). They will not respect any other criteria that the initial query has been filtered by.

---

Simple pagination (page number and size) and sorting (directional ordering using standard or custom property attributes) is also specifiable. Defaults are defined for paging (`pageNumber = 1`, `pageSize = 200`) and sorting (`direction = ASC`).

The `QueryCriteria` interface provides a flexible mechanism for specifying different filtering criteria, including and/or composition and a rich set of operators to include:

- 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)
- Standard SQL-92 aggregate functions (SUM, AVG, MIN, MAX, COUNT)

There are four implementations of this interface which can be chained together to define advanced filters.

1. `VaultQueryCriteria` provides filterable criteria on attributes within the Vault states table: status (UNCONSUMED, CONSUMED), state reference(s), contract state type(s), notaries, soft locked states, timestamps (RECORDED, CONSUMED), state constraints (see [Constraint Types](#)), relevancy (ALL, RELEVANT, NON\_RELEVANT).

---

**Note:** Sensible defaults are defined for frequently used attributes (status = UNCONSUMED, always include soft locked states).

---

2. `FungibleAssetQueryCriteria` provides filterable criteria on attributes defined in the Corda Core `FungibleAsset` contract state interface, used to represent assets that are fungible, countable and issued by a specific party (eg. `Cash.State` and `CommodityContract.State` in the Corda finance module). Filterable attributes include: participants(s), owner(s), quantity, issuer party(s) and issuer reference(s).

---

**Note:** All contract states that extend the `FungibleAsset` now automatically persist that interfaces common state attributes to the `vault_fungible_states` table.

---

3. `LinearStateQueryCriteria` provides filterable criteria on attributes defined in the Corda Core `LinearState` and `DealState` contract state interfaces, used to represent entities that continuously supersede themselves, all of which share the same `linearId` (e.g. trade entity states such as the `IRSState` defined in the SIMM valuation demo). Filterable attributes include: participant(s), `linearId(s)`, `uuid(s)`, and `externalId(s)`.

---

**Note:** All contract states that extend `LinearState` or `DealState` now automatically persist those interfaces common state attributes to the `vault_linear_states` table.

---

4. `VaultCustomQueryCriteria` provides the means to specify one or many arbitrary expressions on attributes defined by a custom contract state that implements its own schema as described in the [Persistence](#) documentation and associated examples. Custom criteria expressions are expressed using one of several type-safe `CriteriaExpression`: `BinaryLogical`, `Not`, `ColumnPredicateExpression`, `AggregateFunctionExpression`. The `ColumnPredicateExpression` allows for specification arbitrary criteria using the previously enumerated operator types. The `AggregateFunctionExpression` allows for the specification of an aggregate function type (sum, avg, max, min, count) with optional grouping and sorting. Furthermore, a rich DSL is provided to enable simple construction of custom criteria using any combination of `ColumnPredicate`. See the `Builder` object in `QueryCriteriaUtils` for a complete specification of the DSL.

---

**Note:** Custom contract schemas are automatically registered upon node startup for CorDapps. Please refer to [Persistence](#) for mechanisms of registering custom schemas for different testing purposes.

---

All `QueryCriteria` implementations are composable using `and` and `or` operators.

All `QueryCriteria` implementations provide an explicitly specifiable set of common attributes:

1. State status attribute (`Vault.StateStatus`), which defaults to filtering on UNCONSUMED states. When chaining several criteria using AND / OR, the last value of this attribute will override any previous

2. Contract state types (<Set<Class<out ContractState>>), which will contain at minimum one type (by default this will be ContractState which resolves to all state types). When chaining several criteria using and and or operators, all specified contract state types are combined into a single set

An example of a custom query is illustrated here:

```
val generalCriteria = VaultQueryCriteria(Vault.StateStatus.ALL)

val results = builder {
    val currencyIndex = PersistentCashState::currency.equal(USD.currencyCode)
    val quantityIndex = PersistentCashState::pennies.greaterThanOrEqualTo(10L)

    val customCriteria1 = VaultCustomQueryCriteria(currencyIndex)
    val customCriteria2 = VaultCustomQueryCriteria(quantityIndex)

    val criteria = generalCriteria.and(customCriteria1.and(customCriteria2))
    vaultService.queryBy<Cash.State>(criteria)
}
```

---

**Note:** Custom contract states that implement the `Queryable` interface may now extend common schemas types `FungiblePersistentState` or, `LinearPersistentState`. Previously, all custom contracts extended the root `PersistentState` class and defined repeated mappings of `FungibleAsset` and `LinearState` attributes. See `SampleCashSchemaV2` and `DummyLinearStateSchemaV2` as examples.

---

Examples of these `QueryCriteria` objects are presented below for Kotlin and Java.

---

**Note:** When specifying the `ContractType` as a parameterised type to the `QueryCriteria` in Kotlin, queries now include all concrete implementations of that type if this is an interface. Previously, it was only possible to query on concrete types (or the universe of all `ContractState`).

---

The Vault Query API leverages the rich semantics of the underlying JPA Hibernate based `Persistence` framework adopted by Corda.

---

**Note:** Permissioning at the database level will be enforced at a later date to ensure authenticated, role-based, read-only access to underlying Corda tables.

---

---

**Note:** API's now provide ease of use calling semantics from both Java and Kotlin. However, it should be noted that Java custom queries are significantly more verbose due to the use of reflection fields to reference schema attribute types.

---

An example of a custom query in Java is illustrated here:

```
QueryCriteria generalCriteria = new VaultQueryCriteria(Vault.StateStatus.ALL);

FieldInfo attributeCurrency = getField("currency", CashSchemaV1.PersistentCashState.
    ↪class);
FieldInfo attributeQuantity = getField("pennies", CashSchemaV1.PersistentCashState.
    ↪class);

CriteriaExpression currencyIndex = Builder.equal(attributeCurrency, "USD");
CriteriaExpression quantityIndex = Builder.greaterThanOrEqualTo(attributeQuantity, 10L);
```

(continues on next page)

(continued from previous page)

```
QueryCriteria customCriteria2 = new VaultCustomQueryCriteria(quantityIndex);
QueryCriteria customCriteria1 = new VaultCustomQueryCriteria(currencyIndex);

QueryCriteria criteria = generalCriteria.and(customCriteria1).and(customCriteria2);
Vault.Page<ContractState> results = vaultService.queryBy(Cash.State.class, criteria);
```

**Note:** Queries by Party specify the AbstractParty which may be concrete or anonymous. In the later case, where an anonymous party does not resolve to an X500 name via the IdentityService, no query results will ever be produced. For performance reasons, queries do not use PublicKey as search criteria.

Custom queries can be either case sensitive or case insensitive. They are defined via a Boolean as one of the function parameters of each operator function. By default each operator is case sensitive.

An example of a case sensitive custom query operator is illustrated here:

```
val currencyIndex = PersistentCashState::currency.equal(USD.currencyCode, true)
```

**Note:** The Boolean input of true in this example could be removed since the function will default to true when not provided.

An example of a case insensitive custom query operator is illustrated here:

```
val currencyIndex = PersistentCashState::currency.equal(USD.currencyCode, false)
```

An example of a case sensitive custom query operator in Java is illustrated here:

```
FieldInfo attributeCurrency = getField("currency", CashSchemaV1.PersistentCashState.
    ↪class);
CriteriaExpression currencyIndex = Builder.equal(attributeCurrency, "USD", true);
```

An example of a case insensitive custom query operator in Java is illustrated here:

```
FieldInfo attributeCurrency = getField("currency", CashSchemaV1.PersistentCashState.
    ↪class);
CriteriaExpression currencyIndex = Builder.equal(attributeCurrency, "USD", false);
```

## 4.5.2 Pagination

The API provides support for paging where large numbers of results are expected (by default, a page size is set to 200 results). Defining a sensible default page size enables efficient checkpointing within flows, and frees the developer from worrying about pagination where result sets are expected to be constrained to 200 or fewer entries. Where large result sets are expected (such as using the RPC API for reporting and/or UI display), it is strongly recommended to define a PageSpecification to correctly process results with efficient memory utilisation. A fail-fast mode is in place to alert API users to the need for pagination where a single query returns more than 200 results and no PageSpecification has been supplied.

Here's a query that extracts every unconsumed ContractState from the vault in pages of size 200, starting from the default page number (page one):

```
val vaultSnapshot = proxy.vaultQueryBy<ContractState>(
    QueryCriteria.VaultQueryCriteria(Vault.StateStatus.UNCONSUMED),
    PageSpecification(DEFAULT_PAGE_NUM, 200))
```

---

**Note:** A pages maximum size MAX\_PAGE\_SIZE is defined as Int.MAX\_VALUE and should be used with extreme caution as results returned may exceed your JVM's memory footprint.

---

### 4.5.3 Example usage

#### Kotlin

**General snapshot queries using VaultQueryCriteria:**

Query for all unconsumed states (simplest query possible):

```
val result = vaultService.queryBy<ContractState>()

/**
 * Query result returns a [Vault.Page] which contains:
 * 1) actual states as a list of [StateAndRef]
 * 2) state reference and associated vault metadata as a list of [Vault.
 *   ↪StateMetadata]
 * 3) [PageSpecification] used to delimit the size of items returned in the result
 *   ↪set (defaults to [DEFAULT_PAGE_SIZE])
 * 4) Total number of items available (to aid further pagination if required)
 */
val states = result.states
val metadata = result.statesMetadata
```

Query for unconsumed states for some state references:

```
val sortAttribute = SortAttribute.Standard(Sort.CommonStateAttribute.STATE_REF_TXN_ID)
val criteria = VaultQueryCriteria(stateRefs = listOf(stateRefs.first(), stateRefs.
    ↪last()))
val results = vaultService.queryBy<DummyLinearContract.State>(criteria,
    ↪Sort(setOf(Sort.SortColumn(sortAttribute, Sort.Direction.ASC))))
```

Query for unconsumed states for several contract state types:

```
val criteria = VaultQueryCriteria(contractStateTypes = setOf(Cash.State::class.java,
    ↪DealState::class.java))
val results = vaultService.queryBy<ContractState>(criteria)
```

Query for unconsumed states for specified contract state constraint types and sorted in ascending alphabetical order:

```
val constraintTypeCriteria = VaultQueryCriteria(constraintTypes = setOf(HASH, CZ_
    ↪WHITELISTED))
val sortAttribute = SortAttribute.Standard(Sort.VaultStateAttribute.CONSTRAINT_TYPE)
val sorter = Sort(setOf(Sort.SortColumn(sortAttribute, Sort.Direction.ASC)))
val constraintResults = vaultService.queryBy<LinearState>(constraintTypeCriteria,
    ↪sorter)
```

Query for unconsumed states for specified contract state constraints (type and data):

```
val constraintCriteria = VaultQueryCriteria(constraints = setOf(Vault.  
    ConstraintInfo(constraintSignature),  
    Vault.ConstraintInfo(constraintSignatureCompositeKey), Vault.  
    ConstraintInfo(constraintHash)))  
val constraintResults = vaultService.queryBy<LinearState>(constraintCriteria)
```

Query for unconsumed states for a given notary:

```
val criteria = VaultQueryCriteria(notary = listOf(CASH_NOTARY))  
val results = vaultService.queryBy<ContractState>(criteria)
```

Query for unconsumed states for a given set of participants:

```
val criteria = LinearStateQueryCriteria(participants = listOf(BIG_CORP, MINI_CORP))  
val results = vaultService.queryBy<ContractState>(criteria)
```

Query for unconsumed states recorded between two time intervals:

```
val start = TODAY  
val end = TODAY.plus(30, ChronoUnit.DAYS)  
val recordedBetweenExpression = TimeCondition(  
    QueryCriteria.TimeInstantType.RECORDED,  
    ColumnPredicate.Between(start, end))  
val criteria = VaultQueryCriteria(timeCondition = recordedBetweenExpression)  
val results = vaultService.queryBy<ContractState>(criteria)
```

---

**Note:** This example illustrates usage of a Between ColumnPredicate.

Query for all states with pagination specification (10 results per page):

```
val pagingSpec = PageSpecification(DEFAULT_PAGE_NUM, 10)  
val criteria = VaultQueryCriteria(status = Vault.StateStatus.ALL)  
val results = vaultService.queryBy<ContractState>(criteria, paging = pagingSpec)
```

---

**Note:** The result set metadata field *totalStatesAvailable* allows you to further paginate accordingly as demonstrated in the following example.

Query for all states using a pagination specification and iterate using the *totalStatesAvailable* field until no further pages available:

```
var pageNumber = DEFAULT_PAGE_NUM  
val states = mutableListOf<StateAndRef<ContractState>>()  
do {  
    val pageSpec = PageSpecification(pageNumber = pageNumber, pageSize = pageSize)  
    val results = vaultService.queryBy<ContractState>(VaultQueryCriteria(), pageSpec)  
    states.addAll(results.states)  
    pageNumber++  
} while ((pageSpec.pageSize * (pageNumber - 1)) <= results.totalStatesAvailable)
```

Query for only relevant states in the vault:

```
    val relevancyAllCriteria = VaultQueryCriteria(relevancyStatus = Vault.  
        ↪RelevancyStatus.RELEVANT)  
    val allDealStateCount = vaultService.queryBy<DummyDealContract.State>  
        ↪(relevancyAllCriteria).states
```

### LinearState and DealState queries using LinearStateQueryCriteria:

Query for unconsumed linear states for given linear ids:

```
val linearIds = issuedStates.states.map { it.state.data.linearId }.toList()  
val criteria = LinearStateQueryCriteria(linearId = listOf(linearIds.first(),  
    ↪linearIds.last()))  
val results = vaultService.queryBy<LinearState>(criteria)
```

Query for all linear states associated with a linear id:

```
val linearStateCriteria = LinearStateQueryCriteria(linearId = listOf(linearId),  
    ↪status = Vault.StateStatus.ALL)  
val vaultCriteria = VaultQueryCriteria(status = Vault.StateStatus.ALL)  
val results = vaultService.queryBy<LinearState>(linearStateCriteria and vaultCriteria)
```

Query for unconsumed deal states with deals references:

```
val criteria = LinearStateQueryCriteria(externalId = listOf("456", "789"))  
val results = vaultService.queryBy<DealState>(criteria)
```

Query for unconsumed deal states with deals parties:

```
val criteria = LinearStateQueryCriteria(participants = parties)  
val results = vaultService.queryBy<DealState>(criteria)
```

Query for only relevant linear states in the vault:

```
    val allLinearStateCriteria = LinearStateQueryCriteria(relevancyStatus = Vault.  
        ↪RelevancyStatus.RELEVANT)  
    val allLinearStates = vaultService.queryBy<DummyLinearContract.State>  
        ↪(allLinearStateCriteria).states
```

### FungibleAsset and DealState queries using FungibleAssetQueryCriteria:

Query for fungible assets for a given currency:

```
val ccyIndex = builder { CashSchemaV1.PersistentCashState::currency.equal(USD.  
    ↪currencyCode) }  
val criteria = VaultCustomQueryCriteria(ccyIndex)  
val results = vaultService.queryBy<FungibleAsset<*>>(criteria)
```

Query for fungible assets for a minimum quantity:

```
val fungibleAssetCriteria = FungibleAssetQueryCriteria(quantity = builder {  
    ↪greaterThan(2500L) })  
val results = vaultService.queryBy<Cash.State>(fungibleAssetCriteria)
```

---

**Note:** This example uses the builder DSL.

---

Query for fungible assets for a specific issuer party:

```
val criteria = FungibleAssetQueryCriteria(issuer = listOf(BOC))
val results = vaultService.queryBy<FungibleAsset<*>>(criteria)
```

Query for only relevant fungible states in the vault:

```
val allCashCriteria = FungibleStateQueryCriteria(relevancyStatus = Vault.
    ↪RelevancyStatus.RELEVANT)
val allCashStates = vaultService.queryBy<Cash.State>(allCashCriteria).states
```

**Aggregate Function queries using VaultCustomQueryCriteria:**

---

**Note:** Query results for aggregate functions are contained in the `otherResults` attribute of a results Page.

---

Aggregations on cash using various functions:

```
val sum = builder { CashSchemaV1.PersistentCashState::pennies.sum() }
val sumCriteria = VaultCustomQueryCriteria(sum)

val count = builder { CashSchemaV1.PersistentCashState::pennies.count() }
val countCriteria = VaultCustomQueryCriteria(count)

val max = builder { CashSchemaV1.PersistentCashState::pennies.max() }
val maxCriteria = VaultCustomQueryCriteria(max)

val min = builder { CashSchemaV1.PersistentCashState::pennies.min() }
val minCriteria = VaultCustomQueryCriteria(min)

val avg = builder { CashSchemaV1.PersistentCashState::pennies.avg() }
val avgCriteria = VaultCustomQueryCriteria(avg)

val results = vaultService.queryBy<FungibleAsset<*>>(sumCriteria
    .and(countCriteria)
    .and(maxCriteria)
    .and(minCriteria)
    .and(avgCriteria))
```

---

**Note:** `otherResults` will contain 5 items, one per calculated aggregate function.

---

Aggregations on cash grouped by currency for various functions:

```
val sum = builder { CashSchemaV1.PersistentCashState::pennies.sum(groupByColumns =
    ↪listOf(CashSchemaV1.PersistentCashState::currency)) }
val sumCriteria = VaultCustomQueryCriteria(sum)

val max = builder { CashSchemaV1.PersistentCashState::pennies.max(groupByColumns =
    ↪listOf(CashSchemaV1.PersistentCashState::currency)) }
val maxCriteria = VaultCustomQueryCriteria(max)

val min = builder { CashSchemaV1.PersistentCashState::pennies.min(groupByColumns =
    ↪listOf(CashSchemaV1.PersistentCashState::currency)) }
val minCriteria = VaultCustomQueryCriteria(min)

val avg = builder { CashSchemaV1.PersistentCashState::pennies.avg(groupByColumns =
    ↪listOf(CashSchemaV1.PersistentCashState::currency)) }
```

(continues on next page)

(continued from previous page)

```
val avgCriteria = VaultCustomQueryCriteria(avg)

val results = vaultService.queryBy<FungibleAsset<>>(sumCriteria
    .and(maxCriteria)
    .and(minCriteria)
    .and(avgCriteria))
```

**Note:** otherResults will contain 24 items, one result per calculated aggregate function per currency (the grouping attribute - currency in this case - is returned per aggregate result).

Sum aggregation on cash grouped by issuer party and currency and sorted by sum:

```
val sum = builder {
    CashSchemaV1.PersistentCashState::pennies.sum(groupByColumns =
        listOf(CashSchemaV1.PersistentCashState::issuerPartyHash,
            CashSchemaV1.PersistentCashState::currency),
        orderBy = Sort.Direction.DESC)
}

val results = vaultService.queryBy<FungibleAsset<>>(VaultCustomQueryCriteria(sum))
```

**Note:** otherResults will contain 12 items sorted from largest summed cash amount to smallest, one result per calculated aggregate function per issuer party and currency (grouping attributes are returned per aggregate result).

Dynamic queries (also using `VaultQueryCriteria`) are an extension to the snapshot queries by returning an additional `QueryResults` return type in the form of an `Observable<Vault.Update>`. Refer to [ReactiveX Observable](#) for a detailed understanding and usage of this type.

Track unconsumed cash states:

```
vaultService.trackBy<Cash.State>().updates // UNCONSUMED default
```

Track unconsumed linear states:

```
val (snapshot, updates) = vaultService.trackBy<LinearState>()
```

**Note:** This will return both `DealState` and `LinearState` states.

Track unconsumed deal states:

```
val (snapshot, updates) = vaultService.trackBy<DealState>()
```

**Note:** This will return only `DealState` states.

## Java examples

Query for all unconsumed linear states:

```
Vault.Page<LinearState> results = vaultService.queryBy(LinearState.class);
```

Query for all consumed cash states:

```
VaultQueryCriteria criteria = new VaultQueryCriteria(Vault.StateStatus.CONSUMED);
Vault.Page<Cash.State> results = vaultService.queryBy(Cash.State.class, criteria);
```

Query for consumed deal states or linear ids, specify a paging specification and sort by unique identifier:

```
Vault.StateStatus status = Vault.StateStatus.CONSUMED;
@SuppressWarnings("unchecked")
Set<Class<LinearState>> contractStateTypes = new HashSet.singletonList(LinearState.
    ↪class));

QueryCriteria vaultCriteria = new VaultQueryCriteria(status, contractStateTypes);

List<UniqueIdentifier> linearIds = singletonList(ids.getSecond());
QueryCriteria linearCriteriaAll = new LinearStateQueryCriteria(null, linearIds, Vault.
    ↪StateStatus.UNCONSUMED, null);
QueryCriteria dealCriteriaAll = new LinearStateQueryCriteria(null, null, dealIds);

QueryCriteria compositeCriteria1 = dealCriteriaAll.or(linearCriteriaAll);
QueryCriteria compositeCriteria2 = compositeCriteria1.and(vaultCriteria);

PageSpecification pageSpec = new PageSpecification(DEFAULT_PAGE_NUM, MAX_PAGE_SIZE);
Sort.SortColumn sortByUid = new Sort.SortColumn(new SortAttribute.Standard(Sort.
    ↪LinearStateAttribute.UUID), Sort.Direction.DESC);
Sort sorting = new Sort(ImmutableSet.of(sortByUid));
Vault.Page<LinearState> results = vaultService.queryBy(LinearState.class,
    ↪compositeCriteria2, pageSpec, sorting);
```

Query for all states using a pagination specification and iterate using the *totalStatesAvailable* field until no further pages available:

```
int pageNumber = DEFAULT_PAGE_NUM;
List<StateAndRef<Cash.State>> states = new ArrayList<>();
long totalResults;
do {
    PageSpecification pageSpec = new PageSpecification(pageNumber, pageSize);
    Vault.Page<Cash.State> results = vaultService.queryBy(Cash.State.class, new_
        ↪VaultQueryCriteria(), pageSpec);
    totalResults = results.getTotalStatesAvailable();
    List<StateAndRef<Cash.State>> newStates = results.getStates();
    System.out.println(newStates.size());
    states.addAll(results.getStates());
    pageNumber++;
} while ((pageSize * (pageNumber - 1) <= totalResults));
```

### Aggregate Function queries using VaultCustomQueryCriteria:

Aggregations on cash using various functions:

```
FieldInfo pennies = getField("pennies", CashSchemaV1.PersistentCashState.class);

QueryCriteria sumCriteria = new VaultCustomQueryCriteria(sum(pennies));
QueryCriteria countCriteria = new VaultCustomQueryCriteria(Builder.count(pennies));
QueryCriteria maxCriteria = new VaultCustomQueryCriteria(Builder.max(pennies));
```

(continues on next page)

(continued from previous page)

```
QueryCriteria minCriteria = new VaultCustomQueryCriteria(Builder.min(pennies));
QueryCriteria avgCriteria = new VaultCustomQueryCriteria(Builder.avg(pennies));

QueryCriteria criteria = sumCriteria.and(countCriteria).and(maxCriteria).
    ↪and(minCriteria).and(avgCriteria);
Vault.Page<Cash.State> results = vaultService.queryBy(Cash.State.class, criteria);
```

Aggregations on cash grouped by currency for various functions:

```
FieldInfo pennies = getField("pennies", CashSchemaV1.PersistentCashState.class);
FieldInfo currency = getField("currency", CashSchemaV1.PersistentCashState.class);

QueryCriteria sumCriteria = new VaultCustomQueryCriteria(sum(pennies,
    ↪singletonList(currency)));
QueryCriteria countCriteria = new VaultCustomQueryCriteria(Builder.count(pennies));
QueryCriteria maxCriteria = new VaultCustomQueryCriteria(Builder.max(pennies,
    ↪singletonList(currency)));
QueryCriteria minCriteria = new VaultCustomQueryCriteria(Builder.min(pennies,
    ↪singletonList(currency)));
QueryCriteria avgCriteria = new VaultCustomQueryCriteria(Builder.avg(pennies,
    ↪singletonList(currency)));

QueryCriteria criteria = sumCriteria.and(countCriteria).and(maxCriteria).
    ↪and(minCriteria).and(avgCriteria);
Vault.Page<Cash.State> results = vaultService.queryBy(Cash.State.class, criteria);
```

Sum aggregation on cash grouped by issuer party and currency and sorted by sum:

```
FieldInfo pennies = getField("pennies", CashSchemaV1.PersistentCashState.class);
FieldInfo currency = getField("currency", CashSchemaV1.PersistentCashState.class);
FieldInfo issuerPartyHash = getField("issuerPartyHash", CashSchemaV1.
    ↪PersistentCashState.class);
QueryCriteria sumCriteria = new VaultCustomQueryCriteria(sum(pennies,
    ↪asList(issuerPartyHash, currency), Sort.Direction.DESC));
Vault.Page<Cash.State> results = vaultService.queryBy(Cash.State.class, sumCriteria);
```

Track unconsumed cash states:

```
@SuppressWarnings("unchecked")
Set<Class<ContractState>> contractStateTypes = new HashSet(singletonList(Cash.State.
    ↪class));

VaultQueryCriteria criteria = new VaultQueryCriteria(Vault.StateStatus.UNCONSUMED,
    ↪contractStateTypes);
DataFeed<Vault.Page<ContractState>, Vault.Update<ContractState>> results =
    ↪vaultService.trackBy(ContractState.class, criteria);

Vault.Page<ContractState> snapshot = results.getSnapshot();
```

Track unconsumed deal states or linear states (with snapshot including specification of paging and sorting by unique identifier):

```
@SuppressWarnings("unchecked")
Set<Class<ContractState>> contractStateTypes = new HashSet(asList(DealState.class,
    ↪LinearState.class));
QueryCriteria vaultCriteria = new VaultQueryCriteria(Vault.StateStatus.UNCONSUMED,
    ↪contractStateTypes);
```

(continues on next page)

(continued from previous page)

```

List<UniqueIdentifier> linearIds = singletonList(uid);
List<AbstractParty> dealParty = singletonList(MEGA_CORP.getParty());
QueryCriteria dealCriteria = new LinearStateQueryCriteria(dealParty, null, dealIds);
QueryCriteria linearCriteria = new LinearStateQueryCriteria(dealParty, linearIds, ↳
    ↳ Vault.StateStatus.UNCONSUMED, null);
QueryCriteria dealOrLinearIdCriteria = dealCriteria.or(linearCriteria);
QueryCriteria compositeCriteria = dealOrLinearIdCriteria.and(vaultCriteria);

PageSpecification pageSpec = new PageSpecification(DEFAULT_PAGE_NUM, MAX_PAGE_SIZE);
Sort.SortColumn sortByUid = new Sort.SortColumn(new SortAttribute.Standard(Sort.
    ↳ LinearStateAttribute.UUID), Sort.Direction.DESC);
Sort sorting = new Sort(ImmutableSet.of(sortByUid));
DataFeed<Vault.Page<ContractState>, Vault.Update<ContractState>> results = ↳
    ↳ vaultService.trackBy(ContractState.class, compositeCriteria, pageSpec, sorting);

Vault.Page<ContractState> snapshot = results.getSnapshot();

```

#### 4.5.4 Troubleshooting

If the results you were expecting do not match actual returned query results we recommend you add an entry to your `log4j2.xml` configuration file to enable display of executed SQL statements:

```

<Logger name="org.hibernate.SQL" level="debug" additivity="false">
    <AppenderRef ref="Console-Appender"/>
</Logger>

```

#### 4.5.5 Behavioural notes

1. TrackBy updates do not take into account the full criteria specification due to different and more restrictive syntax in `observables` filtering (vs full SQL-92 JDBC filtering as used in snapshot views). Specifically, dynamic updates are filtered by `contractStateType` and `stateType` (UNCONSUMED, CONSUMED, ALL) only
2. QueryBy and TrackBy snapshot views using pagination may return different result sets as each paging request is a separate SQL query on the underlying database, and it is entirely conceivable that state modifications are taking place in between and/or in parallel to paging requests. When using pagination, always check the value of the `totalStatesAvailable` (from the `Vault.Page` result) and adjust further paging requests appropriately.

#### 4.5.6 Other use case scenarios

For advanced use cases that require sophisticated pagination, sorting, grouping, and aggregation functions, it is recommended that the CorDapp developer utilise one of the many proven frameworks that ship with this capability out of the box. Namely, implementations of JPQL (JPA Query Language) such as Hibernate for advanced SQL access, and Spring Data for advanced pagination and ordering constructs.

The Corda Tutorials provide examples satisfying these additional Use Cases:

1. Example CorDapp service using Vault API Custom Query to access attributes of IOU State

2. Example CorDapp service query extension executing Named Queries via [JPQL](#)
3. Advanced pagination queries using Spring Data [JPA](#)

#### 4.5.7 Mapping owning keys to external IDs

When creating new public keys via the `KeyManagementService`, it is possible to create an association between the newly created public key and an external ID. This, in effect, allows CorDapp developers to group state ownership/participation keys by an account ID.

---

**Note:** This only works with freshly generated public keys and *not* the node's legal identity key. If you require that the freshly generated keys be for the node's identity then use `PersistentKeyManagementService.freshKeyAndCert` instead of `freshKey`. Currently, the generation of keys for other identities is not supported.

---

The code snippet below show how keys can be associated with an external ID by using the exposed JPA functionality:

```
public AnonymousParty freshKeyForExternalId(UUID externalId, ServiceHub services) {  
    // Create a fresh key pair and return the public key.  
    AnonymousParty anonymousParty = freshKey();  
    // Associate the fresh key to an external ID.  
    services.withEntityManager(entityManager -> {  
        PersistentKeyManagementService.PublicKeyHashToExternalId mapping =  
        ↪PersistentKeyManagementService.PublicKeyHashToExternalId(externalId, anonymousParty.  
        ↪owningKey);  
        entityManager.persist(mapping);  
        return null;  
    });  
    return anonymousParty;  
}
```

```
fun freshKeyForExternalId(externalId: UUID, services: ServiceHub): AnonymousParty {  
    // Create a fresh key pair and return the public key.  
    val anonymousParty = freshKey();  
    // Associate the fresh key to an external ID.  
    services.withEntityManager {  
        val mapping = PersistentKeyManagementService.  
        ↪PublicKeyHashToExternalId(externalId, anonymousParty.owningKey)  
        persist(mapping)  
    }  
    return anonymousParty  
}
```

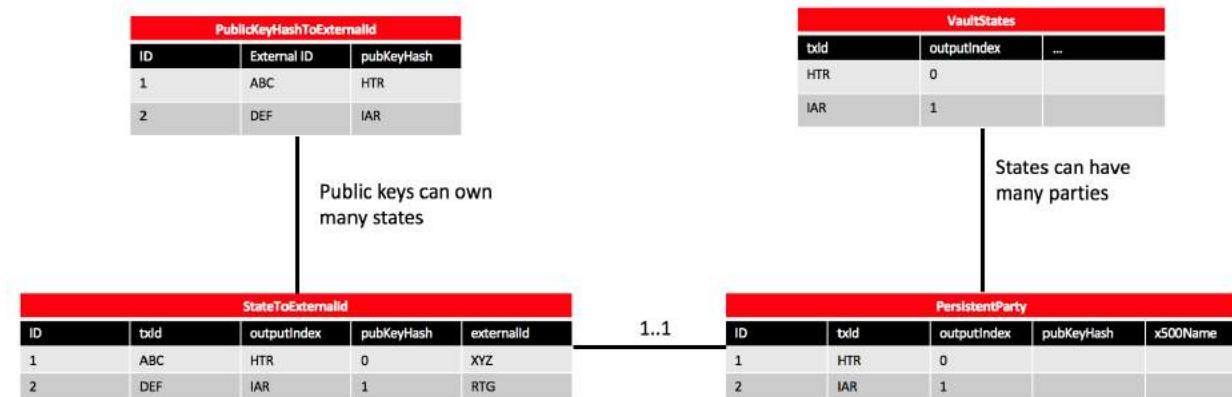
As can be seen in the code snippet above, the `PublicKeyHashToExternalId` entity has been added to `PersistentKeyManagementService`, which allows you to associate your public keys with external IDs. So far, so good.

---

**Note:** Here, it is worth noting that we must map **owning keys** to external IDs, as opposed to **state objects**. This is because it might be the case that a `LinearState` is owned by two public keys generated by the same node.

---

The intuition here is that when these public keys are used to own or participate in a state object, it is trivial to then associate those states with a particular external ID. Behind the scenes, when states are persisted to the vault, the owning keys for each state are persisted to a `PersistentParty` table. The `PersistentParty` table can be joined with the `PublicKeyHashToExternalId` table to create a view which maps each state to one or more external IDs. The entity relationship diagram below helps to explain how this works.



When performing a vault query, it is now possible to query for states by external ID using a custom query criteria.

```
UUID id = someExternalId;
FieldInfo externalIdField = getField("externalId", VaultSchemaV1.StateToExternalId.
    →class);
CriteriaExpression externalId = Builder.equal(externalIdField, id);
QueryCriteria query = new VaultCustomQueryCriteria(externalId);
Vault.Page<StateType> results = vaultService.queryBy(StateType.class, query);
```

```
val id: UUID = someExternalId
val externalId = builder { VaultSchemaV1.StateToExternalId::externalId.equal(id) }
val queryCriteria = QueryCriteria.VaultCustomQueryCriteria(externalId)
val results = vaultService.queryBy<StateType>(queryCriteria).states
```

The `VaultCustomQueryCriteria` can also be combined with other query criteria, like custom schemas, for instance. See the vault query API examples above for how to combine `QueryCriteria`.

## 4.6 API: Transactions

---

**Note:** Before reading this page, you should be familiar with the key concepts of *Transactions*.

---

### Contents

- *API: Transactions*
  - *Transaction lifecycle*
  - *Transaction components*
    - \* *Input states*
      - *Reference input states*
    - \* *Output states*
    - \* *Commands*
    - \* *Attachments*

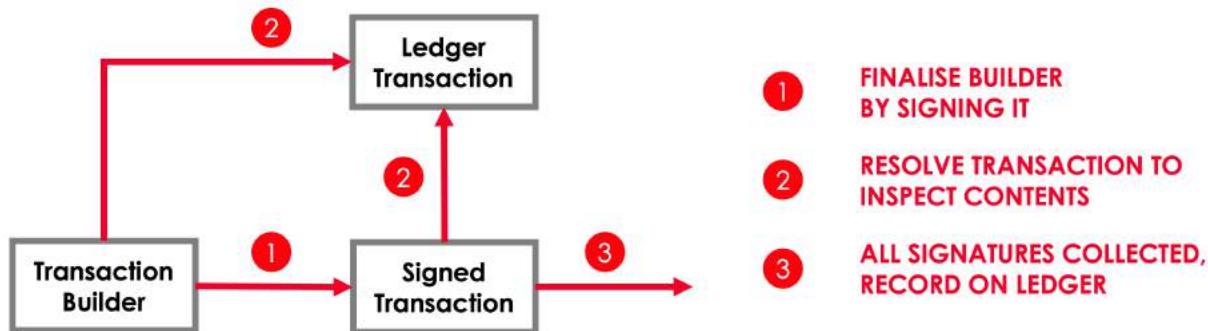
- \* *Time-windows*
- *TransactionBuilder*
  - \* *Creating a builder*
  - \* *Adding items*
  - \* *Signing the builder*
- *SignedTransaction*
  - \* *Verifying the transaction's contents*
  - \* *Verifying the transaction's signatures*
  - \* *Signing the transaction*
  - \* *Notarising and recording*

#### 4.6.1 Transaction lifecycle

Between its creation and its final inclusion on the ledger, a transaction will generally occupy one of three states:

- *TransactionBuilder*. A transaction's initial state. This is the only state during which the transaction is mutable, so we must add all the required components before moving on.
- *SignedTransaction*. The transaction now has one or more digital signatures, making it immutable. This is the transaction type that is passed around to collect additional signatures and that is recorded on the ledger.
- *LedgerTransaction*. The transaction has been “resolved” - for example, its inputs have been converted from references to actual states - allowing the transaction to be fully inspected.

We can visualise the transitions between the three stages as follows:



#### 4.6.2 Transaction components

A transaction consists of six types of components:

- 1+ states:
  - 0+ input states
  - 0+ output states
  - 0+ reference input states

- 1+ commands
- 0+ attachments
- 0 or 1 time-window
  - A transaction with a time-window must also have a notary

Each component corresponds to a specific class in the Corda API. The following section describes each component class, and how it is created.

## Input states

An input state is added to a transaction as a `StateAndRef`, which combines:

- The `ContractState` itself
- A `StateRef` identifying this `ContractState` as the output of a specific transaction

```
val ourStateAndRef: StateAndRef<DummyState> = serviceHub.toStateAndRef<DummyState>
    ↪(ourStateRef)
```

```
StateAndRef ourStateAndRef = getServiceHub().toStateAndRef(ourStateRef);
```

A `StateRef` uniquely identifies an input state, allowing the notary to mark it as historic. It is made up of:

- The hash of the transaction that generated the state
- The state's index in the outputs of that transaction

```
val ourStateRef: StateRef = StateRef(SecureHash.sha256("DummyTransactionHash"), 0)
```

```
StateRef ourStateRef = new StateRef(SecureHash.sha256("DummyTransactionHash"), 0);
```

The `StateRef` links an input state back to the transaction that created it. This means that transactions form “chains” linking each input back to an original issuance transaction. This allows nodes verifying the transaction to “walk the chain” and verify that each input was generated through a valid sequence of transactions.

## Reference input states

**Warning:** Reference states are only available on Corda networks with a minimum platform version >= 4.

A reference input state is added to a transaction as a `ReferencedStateAndRef`. A `ReferencedStateAndRef` can be obtained from a `StateAndRef` by calling the `StateAndRef.referenced()` method which returns a `ReferencedStateAndRef`.

```
val referenceState: ReferencedStateAndRef<DummyState> = ourStateAndRef.referenced()
```

```
ReferencedStateAndRef referenceState = ourStateAndRef.referenced();
```

### Handling of update races:

When using reference states in a transaction, it may be the case that a notarisation failure occurs. This is most likely because the creator of the state (being used as a reference state in your transaction), has just updated it.

Typically, the creator of such reference data will have implemented flows for syndicating the updates out to users. However it is inevitable that there will be a delay between the state being used as a reference being consumed, and the nodes using it receiving the update.

This is where the `WithReferencedStatesFlow` comes in. Given a flow which uses reference states, the `WithReferencedStatesFlow` will execute the the flow as a subFlow. If the flow fails due to a `NotaryError`. Conflict for a reference state, then it will be suspended until the state refs for the reference states are consumed. In this case, a consumption means that:

1. the owner of the reference state has updated the state with a valid, notarised transaction
2. the owner of the reference state has shared the update with the node attempting to run the flow which uses the reference state
3. The node has successfully committed the transaction updating the reference state (and all the dependencies), and added the updated reference state to the vault.

At the point where the transaction updating the state being used as a reference is committed to storage and the vault update occurs, then the `WithReferencedStatesFlow` will wake up and re-execute the provided flow.

**Warning:** Caution should be taken when using this flow as it facilitates automated re-running of flows which use reference states. The flow using reference states should include checks to ensure that the reference data is reasonable, especially if the economics of the transaction depends upon the data contained within a reference state.

## Output states

Since a transaction's output states do not exist until the transaction is committed, they cannot be referenced as the outputs of previous transactions. Instead, we create the desired output states as `ContractState` instances, and add them to the transaction directly:

```
val ourOutputState: DummyState = DummyState()
```

```
DummyState ourOutputState = new DummyState();
```

In cases where an output state represents an update of an input state, we may want to create the output state by basing it on the input state:

```
val ourOtherOutputState: DummyState = ourOutputState.copy(magicNumber = 77)
```

```
DummyState ourOtherOutputState = ourOutputState.copy(77);
```

Before our output state can be added to a transaction, we need to associate it with a contract. We can do this by wrapping the output state in a `StateAndContract`, which combines:

- The `ContractState` representing the output states
- A String identifying the contract governing the state

```
val ourOutput: StateAndContract = StateAndContract(ourOutputState, DummyContract.  
    ↪PROGRAM_ID)
```

```
StateAndContract ourOutput = new StateAndContract(ourOutputState, DummyContract.  
    ↪PROGRAM_ID);
```

## Commands

A command is added to the transaction as a Command, which combines:

- A CommandData instance indicating the command's type
- A List<PublicKey> representing the command's required signers

```
val commandData: DummyContract.Commands.Create = DummyContract.Commands.Create()
val ourPubKey: PublicKey = serviceHub.myInfo.legalIdentitiesAndCerts.first().owningKey
val counterpartyPubKey: PublicKey = counterparty.owningKey
val requiredSigners: List<PublicKey> = listOf(ourPubKey, counterpartyPubKey)
val ourCommand: Command<DummyContract.Commands.Create> = Command(commandData,_
    ↴requiredSigners)
```

```
DummyContract.Commands.Create commandData = new DummyContract.Commands.Create();
PublicKey ourPubKey = getServiceHub().getMyInfo().getLegalIdentitiesAndCerts().get(0).
    ↴getOwningKey();
PublicKey counterpartyPubKey = counterparty.getOwningKey();
List<PublicKey> requiredSigners = ImmutableList.of(ourPubKey, counterpartyPubKey);
Command<DummyContract.Commands.Create> ourCommand = new Command<>(commandData,_
    ↴requiredSigners);
```

## Attachments

Attachments are identified by their hash:

```
val ourAttachment: SecureHash = SecureHash.sha256("DummyAttachment")  
  
SecureHash ourAttachment = SecureHash.sha256("DummyAttachment");
```

The attachment with the corresponding hash must have been uploaded ahead of time via the node's RPC interface.

## Time-windows

Time windows represent the period during which the transaction must be notarised. They can have a start and an end time, or be open at either end:

```
val ourTimeWindow: TimeWindow = TimeWindow.between(Instant.MIN, Instant.MAX)
val ourAfter: TimeWindow = TimeWindow.fromOnly(Instant.MIN)
val ourBefore: TimeWindow = TimeWindow.untilOnly(Instant.MAX)  
  
TimeWindow ourTimeWindow = TimeWindow.between(Instant.MIN, Instant.MAX);
TimeWindow ourAfter = TimeWindow.fromOnly(Instant.MIN);
TimeWindow ourBefore = TimeWindow.untilOnly(Instant.MAX);
```

We can also define a time window as an Instant plus/minus a time tolerance (e.g. 30 seconds):

```
val ourTimeWindow2: TimeWindow = TimeWindow.withTolerance(serviceHub.clock.instant(),_
    ↴30.seconds)  
  
TimeWindow ourTimeWindow2 = TimeWindow.withTolerance(getServiceHub().getClock() .
    ↴instant(), Duration.ofSeconds(30));
```

Or as a start-time plus a duration:

```
val ourTimeWindow3: TimeWindow = TimeWindow.fromStartAndDuration(serviceHub.clock.  
    ↪instant(), 30.seconds)
```

```
TimeWindow ourTimeWindow3 = TimeWindow.fromStartAndDuration(getServiceHub().  
    ↪getClock().instant(), Duration.ofSeconds(30));
```

### 4.6.3 TransactionBuilder

#### Creating a builder

The first step when creating a transaction proposal is to instantiate a `TransactionBuilder`.

If the transaction has input states or a time-window, we need to instantiate the builder with a reference to the notary that will notarise the inputs and verify the time-window:

```
val txBuilder: TransactionBuilder = TransactionBuilder(specificNotary)
```

```
TransactionBuilder txBuilder = new TransactionBuilder(specificNotary);
```

We discuss the selection of a notary in [API: Flows](#).

If the transaction does not have any input states or a time-window, it does not require a notary, and can be instantiated without one:

```
val txBuilderNoNotary: TransactionBuilder = TransactionBuilder()
```

```
TransactionBuilder txBuilderNoNotary = new TransactionBuilder();
```

#### Adding items

The next step is to build up the transaction proposal by adding the desired components.

We can add components to the builder using the `TransactionBuilder.withItems` method:

```
/** A more convenient way to add items to this transaction that calls the add*  
methods for you based on type */  
fun withItems(vararg items: Any) = apply {  
    for (t in items) {  
        when (t) {  
            is StateAndRef<*> -> addInputState(t)  
            is ReferencedStateAndRef<*> -> addReferenceState(t)  
            is SecureHash -> addAttachment(t)  
            is TransactionState<*> -> addOutputState(t)  
            is StateAndContract -> addOutputState(t.state, t.contract)  
            is ContractState -> throw UnsupportedOperationException("Removed as  
↪of V1: please use a StateAndContract instead")  
            is Command<*> -> addCommand(t)  
            is CommandData -> throw IllegalArgumentException("You passed an  
↪instance of CommandData, but that lacks the pubkey. You need to wrap it in a  
↪Command object first.")  
            is TimeWindow -> setTimeWindow(t)  
            is PrivacySalt -> setPrivacySalt(t)  
            else -> throw IllegalArgumentException("Wrong argument type: ${t.  
↪javaClass}")
```

(continues on next page)

(continued from previous page)

```

        }
    }
}
```

`withItems` takes a `vararg` of objects and adds them to the builder based on their type:

- `StateAndRef` objects are added as input states
- `ReferencedStateAndRef` objects are added as reference input states
- `TransactionState` and `StateAndContract` objects are added as output states
  - Both `TransactionState` and `StateAndContract` are wrappers around a `ContractState` output that link the output to a specific contract
- `Command` objects are added as commands
- `SecureHash` objects are added as attachments
- A `TimeWindow` object replaces the transaction's existing `TimeWindow`, if any

Passing in objects of any other type will cause an `IllegalArgumentException` to be thrown.

Here's an example usage of `TransactionBuilder.withItems`:

```

txBuilder.withItems(
    // Inputs, as ``StateAndRef``'s that reference the outputs of previous transactions
    ourStateAndRef,
    // Outputs, as ``StateAndContract``'s
    ourOutput,
    // Commands, as ``Command``'s
    ourCommand,
    // Attachments, as ``SecureHash``'es
    ourAttachment,
    // A time-window, as ``TimeWindow``
    ourTimeWindow
)

```

```

txBuilder.withItems(
    // Inputs, as ``StateAndRef``'s that reference to the outputs of previous transactions
    ourStateAndRef,
    // Outputs, as ``StateAndContract``'s
    ourOutput,
    // Commands, as ``Command``'s
    ourCommand,
    // Attachments, as ``SecureHash``'es
    ourAttachment,
    // A time-window, as ``TimeWindow``
    ourTimeWindow
);

```

There are also individual methods for adding components.

Here are the methods for adding inputs and attachments:

```

txBuilder.addInputState(ourStateAndRef)
txBuilder.addAttachment(ourAttachment)

```

```
txBuilder.addInputState(ourStateAndRef);  
txBuilder.addAttachment(ourAttachment);
```

An output state can be added as a ContractState, contract class name and notary:

```
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID, specificNotary)
```

```
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID, specificNotary);
```

We can also leave the notary field blank, in which case the transaction's default notary is used:

```
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID)
```

```
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID);
```

Or we can add the output state as a TransactionState, which already specifies the output's contract and notary:

```
val txState: TransactionState<DummyState> = TransactionState(ourOutputState,  
    ↪ DummyContract.PROGRAM_ID, specificNotary)
```

```
TransactionState txState = new TransactionState(ourOutputState, DummyContract.PROGRAM_  
    ↪ ID, specificNotary);
```

Commands can be added as a Command:

```
txBuilder.addCommand(ourCommand)
```

```
txBuilder.addCommand(ourCommand);
```

Or as CommandData and a vararg PublicKey:

```
txBuilder.addCommand(commandData, ourPubKey, counterpartyPubKey)
```

```
txBuilder.addCommand(commandData, ourPubKey, counterpartyPubKey);
```

For the time-window, we can set a time-window directly:

```
txBuilder.setTimeWindow(ourTimeWindow)
```

```
txBuilder.setTimeWindow(ourTimeWindow);
```

Or define the time-window as a time plus a duration (e.g. 45 seconds):

```
txBuilder.setTimeWindow(serviceHub.clock.instant(), 45.seconds)
```

```
txBuilder.setTimeWindow(getServiceHub().getClock().instant(), Duration.ofSeconds(45));
```

### Siging the builder

Once the builder is ready, we finalize it by signing it and converting it into a SignedTransaction.

We can either sign with our legal identity key:

```
val onceSignedTx: SignedTransaction = serviceHub.signInitialTransaction(txBuilder)
```

```
SignedTransaction onceSignedTx = getServiceHub().signInitialTransaction(txBuilder);
```

Or we can also choose to use another one of our public keys:

```
val otherIdentity: PartyAndCertificate = serviceHub.keyManagementService.  
    ↪freshKeyAndCert(ourIdentityAndCert, false)  
val onceSignedTx2: SignedTransaction = serviceHub.signInitialTransaction(txBuilder,  
    ↪otherIdentity.owningKey)
```

```
PartyAndCertificate otherIdentity = getServiceHub().getKeyManagementService().  
    ↪freshKeyAndCert(getOurIdentityAndCert(), false);  
SignedTransaction onceSignedTx2 = getServiceHub().signInitialTransaction(txBuilder,  
    ↪otherIdentity.getOwningKey());
```

Either way, the outcome of this process is to create an immutable `SignedTransaction` with our signature over it.

#### 4.6.4 SignedTransaction

A `SignedTransaction` is a combination of:

- An immutable transaction
- A list of signatures over that transaction

```
@KeepForDJVM  
@CordaSerializable  
data class SignedTransaction(val txBits: SerializedBytes<CoreTransaction>,  
    override val sigs: List<TransactionSignature>  
) : TransactionWithSignatures {
```

Before adding our signature to the transaction, we'll want to verify both the transaction's contents and the transaction's signatures.

##### Verifying the transaction's contents

If a transaction has inputs, we need to retrieve all the states in the transaction's dependency chain before we can verify the transaction's contents. This is because the transaction is only valid if its dependency chain is also valid. We do this by requesting any states in the chain that our node doesn't currently have in its local storage from the proposer(s) of the transaction. This process is handled by a built-in flow called `ReceiveTransactionFlow`. See [API: Flows](#) for more details.

We can now verify the transaction's contents to ensure that it satisfies the contracts of all the transaction's input and output states:

```
twiceSignedTx.verify(serviceHub)
```

```
twiceSignedTx.verify(getServiceHub());
```

Checking that the transaction meets the contract constraints is only part of verifying the transaction's contents. We will usually also want to perform our own additional validation of the transaction contents before signing, to ensure that the transaction proposal represents an agreement we wish to enter into.

However, the `SignedTransaction` holds its inputs as `StateRef` instances, and its attachments as `SecureHash` instances, which do not provide enough information to properly validate the transaction's contents. We first need to resolve the `StateRef` and `SecureHash` instances into actual `ContractState` and `Attachment` instances, which we can then inspect.

We achieve this by using the `ServiceHub` to convert the `SignedTransaction` into a `LedgerTransaction`:

```
val ledgerTx: LedgerTransaction = twiceSignedTx.toLedgerTransaction(serviceHub)
```

```
LedgerTransaction ledgerTx = twiceSignedTx.toLedgerTransaction(getServiceHub());
```

We can now perform our additional verification. Here's a simple example:

```
val outputState: DummyState = ledgerTx.outputsOfType<DummyState>().single()
if (outputState.magicNumber == 777) {
    // ``FlowException`` is a special exception type. It will be
    // propagated back to any counterparty flows waiting for a
    // message from this flow, notifying them that the flow has
    // failed.
    throw FlowException("We expected a magic number of 777.")
}
```

```
DummyState outputState = ledgerTx.outputsOfType(DummyState.class).get(0);
if (outputState.getMagicNumber() != 777) {
    // ``FlowException`` is a special exception type. It will be
    // propagated back to any counterparty flows waiting for a
    // message from this flow, notifying them that the flow has
    // failed.
    throw new FlowException("We expected a magic number of 777.");
}
```

### Verifying the transaction's signatures

Aside from verifying that the transaction's contents are valid, we also need to check that the signatures are valid. A valid signature over the hash of the transaction prevents tampering.

We can verify that all the transaction's required signatures are present and valid as follows:

```
fullySignedTx.verifyRequiredSignatures()
```

```
fullySignedTx.verifyRequiredSignatures();
```

However, we'll often want to verify the transaction's existing signatures before all of them have been collected. For this we can use `SignedTransaction.verifySignaturesExcept`, which takes a `Vararg` of the public keys for which the signatures are allowed to be missing:

```
onceSignedTx.verifySignaturesExcept(counterpartyPubKey)
```

```
onceSignedTx.verifySignaturesExcept(counterpartyPubKey);
```

There is also an overload of `SignedTransaction.verifySignaturesExcept`, which takes a `Collection` of the public keys for which the signatures are allowed to be missing:

```
onceSignedTx.verifySignaturesExcept(listOf(counterpartyPubKey))
```

```
onceSignedTx.verifySignaturesExcept(singletonList(counterpartyPubKey));
```

If the transaction is missing any signatures without the corresponding public keys being passed in, a `SignaturesMissingException` is thrown.

We can also choose to simply verify the signatures that are present:

```
twiceSignedTx.checkSignaturesAreValid()
```

```
twiceSignedTx.checkSignaturesAreValid();
```

Be very careful, however - this function neither guarantees that the signatures that are present are required, nor checks whether any signatures are missing.

## **Signing the transaction**

Once we are satisfied with the contents and existing signatures over the transaction, we add our signature to the `SignedTransaction` to indicate that we approve the transaction.

We can sign using our legal identity key, as follows:

```
val twiceSignedTx: SignedTransaction = serviceHub.addSignature(onceSignedTx)
```

```
SignedTransaction twiceSignedTx = getServiceHub().addSignature(onceSignedTx);
```

Or we can choose to sign using another one of our public keys:

```
val twiceSignedTx2: SignedTransaction = serviceHub.addSignature(onceSignedTx, ↵otherIdentity2.owningKey)
```

```
SignedTransaction twiceSignedTx2 = getServiceHub().addSignature(onceSignedTx, ↵otherIdentity2.getOwningKey());
```

We can also generate a signature over the transaction without adding it to the transaction directly.

We can do this with our legal identity key:

```
val sig: TransactionSignature = serviceHub.createSignature(onceSignedTx)
```

```
TransactionSignature sig = getServiceHub().createSignature(onceSignedTx);
```

Or using another one of our public keys:

```
val sig2: TransactionSignature = serviceHub.createSignature(onceSignedTx, ↵otherIdentity2.owningKey)
```

```
TransactionSignature sig2 = getServiceHub().createSignature(onceSignedTx, ↵otherIdentity2.getOwningKey());
```

## **Notarising and recording**

Notarising and recording a transaction is handled by a built-in flow called `FinalityFlow`. See [API: Flows](#) for more details.

## 4.7 API: Flows

---

**Note:** Before reading this page, you should be familiar with the key concepts of [Flows](#).

---

### Contents

- [API: Flows](#)
  - [An example flow](#)
    - \* [Initiator](#)
    - \* [Responder](#)
  - [FlowLogic](#)
  - [FlowLogic annotations](#)
  - [Call](#)
  - [ServiceHub](#)
  - [Common flow tasks](#)
    - \* [Transaction building](#)
    - \* [Extracting states from the vault](#)
    - \* [Retrieving information about other nodes](#)
      - [Notaries](#)
      - [Specific counterparties](#)
    - \* [Communication between parties](#)
      - [InitiateFlow](#)
      - [Send](#)
      - [Receive](#)
      - [SendAndReceive](#)
      - [Counterparty response](#)
    - \* [Why sessions?](#)
    - \* [Porting from the old Party-based API](#)
  - [Subflows](#)
    - \* [Inlined subflows](#)
    - \* [Initiating subflows](#)
      - [Core initiating subflows](#)
    - \* [Library flows](#)
      - [FinalityFlow](#)
      - [CollectSignaturesFlow/SignTransactionFlow](#)
      - [SendTransactionFlow/ReceiveTransactionFlow](#)

- \* *Why inlined subflows?*
- *FlowException*
- *ProgressTracker*
- *HTTP and database calls*
- *Concurrency, Locking and Waiting*
  - \* *Locking*
  - \* *Waiting*

### 4.7.1 An example flow

Before we discuss the API offered by the flow, let's consider what a standard flow may look like.

Imagine a flow for agreeing a basic ledger update between Alice and Bob. This flow will have two sides:

- An `Initiator` side, that will initiate the request to update the ledger
- A `Responder` side, that will respond to the request to update the ledger

#### **Initiator**

In our flow, the Initiator flow class will be doing the majority of the work:

##### *Part 1 - Build the transaction*

1. Choose a notary for the transaction
2. Create a transaction builder
3. Extract any input states from the vault and add them to the builder
4. Create any output states and add them to the builder
5. Add any commands, attachments and time-window to the builder

##### *Part 2 - Sign the transaction*

6. Sign the transaction builder
7. Convert the builder to a signed transaction

##### *Part 3 - Verify the transaction*

8. Verify the transaction by running its contracts

##### *Part 4 - Gather the counterparty's signature*

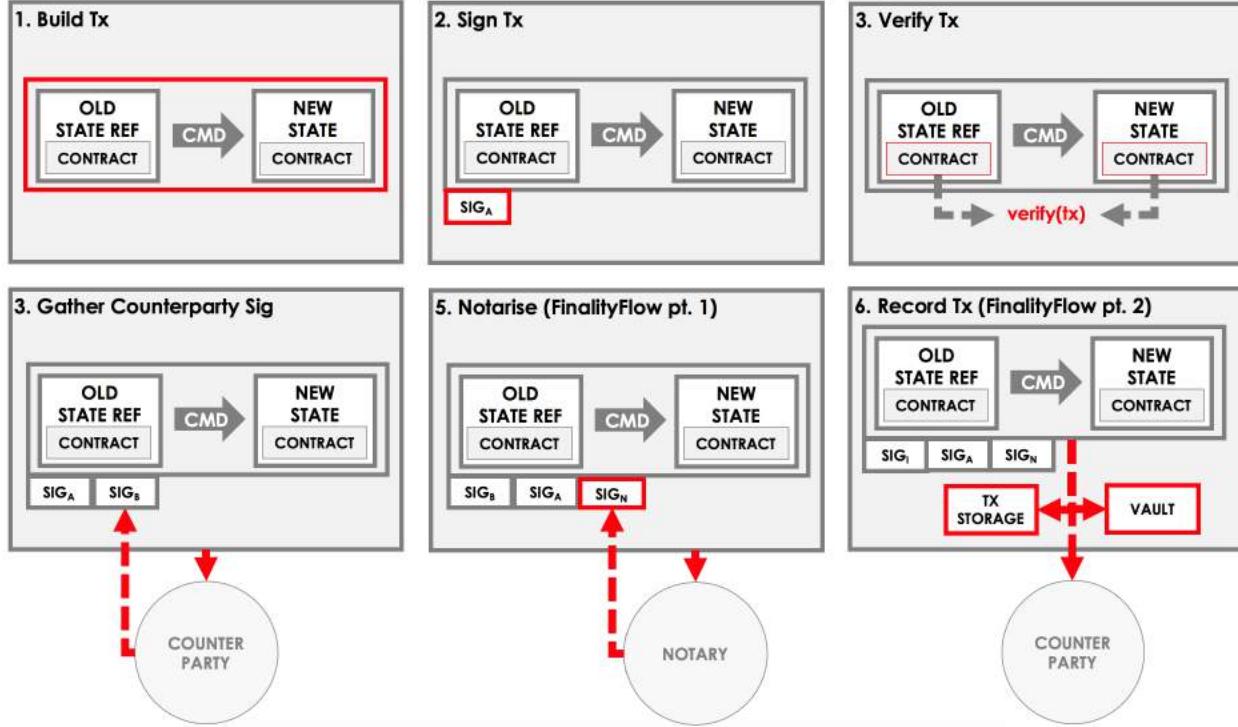
9. Send the transaction to the counterparty
10. Wait to receive back the counterparty's signature
11. Add the counterparty's signature to the transaction
12. Verify the transaction's signatures

##### *Part 5 - Finalize the transaction*

13. Send the transaction to the notary
14. Wait to receive back the notarised transaction

15. Record the transaction locally
16. Store any relevant states in the vault
17. Send the transaction to the counterparty for recording

We can visualize the work performed by initiator as follows:



## Responder

To respond to these actions, the responder takes the following steps:

### *Part 1 - Sign the transaction*

1. Receive the transaction from the counterparty
2. Verify the transaction's existing signatures
3. Verify the transaction by running its contracts
4. Generate a signature over the transaction
5. Send the signature back to the counterparty

### *Part 2 - Record the transaction*

6. Receive the notarised transaction from the counterparty
7. Record the transaction locally
8. Store any relevant states in the vault

## 4.7.2 FlowLogic

In practice, a flow is implemented as one or more communicating FlowLogic subclasses. The FlowLogic subclass's constructor can take any number of arguments of any type. The generic of FlowLogic (e.g. FlowLogic<SignedTransaction>) indicates the flow's return type.

```
class Initiator(val arg1: Boolean,
               val arg2: Int,
               val counterparty: Party): FlowLogic<SignedTransaction>() { }

class Responder(val otherParty: Party) : FlowLogic<Unit>() { }
```

```
public static class Initiator extends FlowLogic<SignedTransaction> {
    private final boolean arg1;
    private final int arg2;
    private final Party counterparty;

    public Initiator(boolean arg1, int arg2, Party counterparty) {
        this.arg1 = arg1;
        this.arg2 = arg2;
        this.counterparty = counterparty;
    }
}

public static class Responder extends FlowLogic<Void> { }
```

## 4.7.3 FlowLogic annotations

Any flow from which you want to initiate other flows must be annotated with the @InitiatingFlow annotation. Additionally, if you wish to start the flow via RPC, you must annotate it with the @StartableByRPC annotation:

```
@InitiatingFlow
@StartableByRPC
class Initiator(): FlowLogic<Unit>() { }
```

```
@InitiatingFlow
@StartableByRPC
public static class Initiator extends FlowLogic<Unit> { }
```

Meanwhile, any flow that responds to a message from another flow must be annotated with the @InitiatedBy annotation. @InitiatedBy takes the class of the flow it is responding to as its single parameter:

```
@InitiatedBy(Initiator::class)
class Responder(val otherSideSession: FlowSession) : FlowLogic<Unit>() { }
```

```
@InitiatedBy(Initiator.class)
public static class Responder extends FlowLogic<Void> { }
```

Additionally, any flow that is started by a SchedulableState must be annotated with the @SchedulableFlow annotation.

## 4.7.4 Call

Each `FlowLogic` subclass must override `FlowLogic.call()`, which describes the actions it will take as part of the flow. For example, the actions of the initiator's side of the flow would be defined in `Initiator.call`, and the actions of the responder's side of the flow would be defined in `Responder.call`.

In order for nodes to be able to run multiple flows concurrently, and to allow flows to survive node upgrades and restarts, flows need to be checkpointable and serializable to disk. This is achieved by marking `FlowLogic.call()`, as well as any function invoked from within `FlowLogic.call()`, with an `@Suspendable` annotation.

```
class Initiator(val counterparty: Party): FlowLogic<Unit>() {  
    @Suspendable  
    override fun call() { }  
}
```

```
public static class InitiatorFlow extends FlowLogic<Void> {  
    private final Party counterparty;  
  
    public Initiator(Party counterparty) {  
        this.counterparty = counterparty;  
    }  
  
    @Suspendable  
    @Override  
    public Void call() throws FlowException { }  
}
```

## 4.7.5 ServiceHub

Within `FlowLogic.call`, the flow developer has access to the node's `ServiceHub`, which provides access to the various services the node provides. We will use the `ServiceHub` extensively in the examples that follow. You can also see [API: ServiceHub](#) for information about the services the `ServiceHub` offers.

## 4.7.6 Common flow tasks

There are a number of common tasks that you will need to perform within `FlowLogic.call` in order to agree ledger updates. This section details the API for common tasks.

### Transaction building

The majority of the work performed during a flow will be to build, verify and sign a transaction. This is covered in [API: Transactions](#).

### Extracting states from the vault

When building a transaction, you'll often need to extract the states you wish to consume from the vault. This is covered in [API: Vault Query](#).

## Retrieving information about other nodes

We can retrieve information about other nodes on the network and the services they offer using ServiceHub.`networkMapCache`.

### Notaries

Remember that a transaction generally needs a notary to:

- Prevent double-spends if the transaction has inputs
- Serve as a timestamping authority if the transaction has a time-window

There are several ways to retrieve a notary from the network map:

```
val notaryName: CordaX500Name = CordaX500Name(
    organisation = "Notary Service",
    locality = "London",
    country = "GB")
val specificNotary: Party = serviceHub.networkMapCache.getNotary(notaryName)!!
// Alternatively, we can pick an arbitrary notary from the notary
// list. However, it is always preferable to specify the notary
// explicitly, as the notary list might change when new notaries are
// introduced, or old ones decommissioned.
val firstNotary: Party = serviceHub.networkMapCache.notaryIdentities.first()
```

```
CordaX500Name notaryName = new CordaX500Name("Notary Service", "London", "GB");
Party specificNotary = getServiceHub().getNetworkMapCache().getNotary(notaryName);
// Alternatively, we can pick an arbitrary notary from the notary
// list. However, it is always preferable to specify the notary
// explicitly, as the notary list might change when new notaries are
// introduced, or old ones decommissioned.
Party firstNotary = getServiceHub().getNetworkMapCache().getNotaryIdentities().get(0);
```

### Specific counterparties

We can also use the network map to retrieve a specific counterparty:

```
val counterpartyName: CordaX500Name = CordaX500Name(
    organisation = "NodeA",
    locality = "London",
    country = "GB")
val namedCounterparty: Party = serviceHub.identityService.
    ↪wellKnownPartyFromX500Name(counterpartyName) ?:
        throw IllegalArgumentException("Couldn't find counterparty for NodeA in_
        ↪identity service")
val keyedCounterparty: Party = serviceHub.identityService.partyFromKey(dummyPubKey) ?:
    throw IllegalArgumentException("Couldn't find counterparty with key:
    ↪$dummyPubKey in identity service")
```

```
CordaX500Name counterPartyName = new CordaX500Name("NodeA", "London", "GB");
Party namedCounterparty = getServiceHub().getIdentityService().
    ↪wellKnownPartyFromX500Name(counterPartyName);
Party keyedCounterparty = getServiceHub().getIdentityService().
    ↪partyFromKey(dummyPubKey);
```

## Communication between parties

In order to create a communication session between your initiator flow and the receiver flow you must call `initiateFlow(party: Party): FlowSession`

`FlowSession` instances in turn provide three functions:

- **`send(payload: Any)`**
  - Sends the payload object
- **`receive(receiveType: Class<R>): R`**
  - Receives an object of type `receiveType`
- **`sendAndReceive(receiveType: Class<R>, payload: Any): R`**
  - Sends the payload object and receives an object of type `receiveType` back

In addition `FlowLogic` provides functions that batch receives:

- `receiveAllMap(sessions: Map<FlowSession, Class<out Any>>): Map<FlowSession, UntrustworthyData<Any>>` Receives from all `FlowSession` objects specified in the passed in map. The received types may differ.
- `receiveAll(receiveType: Class<R>, sessions: List<FlowSession>): List<UntrustworthyData<R>>` Receives from all `FlowSession` objects specified in the passed in list. The received types must be the same.

The batched functions are implemented more efficiently by the flow framework.

## InitiateFlow

`initiateFlow` creates a communication session with the passed in `Party`.

```
val counterpartySession: FlowSession = initiateFlow(counterparty)
```

```
FlowSession counterpartySession = initiateFlow(counterparty);
```

Note that at the time of call to this function no actual communication is done, this is deferred to the first send/receive, at which point the counterparty will either:

1. Ignore the message if they are not registered to respond to messages from this flow.
2. Start the flow they have registered to respond to this flow.

## Send

Once we have a `FlowSession` object we can send arbitrary data to a counterparty:

```
counterpartySession.send(Any())
```

```
counterpartySession.send(new Object());
```

The flow on the other side must eventually reach a corresponding `receive` call to get this message.

## Receive

We can also wait to receive arbitrary data of a specific type from a counterparty. Again, this implies a corresponding send call in the counterparty's flow. A few scenarios:

- We never receive a message back. In the current design, the flow is paused until the node's owner kills the flow.
- Instead of sending a message back, the counterparty throws a `FlowException`. This exception is propagated back to us, and we can use the error message to establish what happened.
- We receive a message back, but it's of the wrong type. In this case, a `FlowException` is thrown.
- We receive back a message of the correct type. All is good.

Upon calling `receive` (or `sendAndReceive`), the `FlowLogic` is suspended until it receives a response.

We receive the data wrapped in an `UntrustworthyData` instance. This is a reminder that the data we receive may not be what it appears to be! We must unwrap the `UntrustworthyData` using a lambda:

```
val packet1: UntrustworthyData<Int> = counterpartySession.receive<Int>()
val int: Int = packet1.unwrap { data ->
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    data
}
```

```
UntrustworthyData<Integer> packet1 = counterpartySession.receive(Integer.class);
Integer integer = packet1.unwrap(data -> {
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    return data;
});
```

We're not limited to sending to and receiving from a single counterparty. A flow can send messages to as many parties as it likes, and each party can invoke a different response flow:

```
val regulatorSession: FlowSession = initiateFlow(regulator)
regulatorSession.send()
val packet3: UntrustworthyData<Any> = regulatorSession.receive<Any>()
```

```
FlowSession regulatorSession = initiateFlow(regulator);
regulatorSession.send(new Object());
UntrustworthyData<Object> packet3 = regulatorSession.receive(Object.class);
```

**Warning:** If you initiate several flows from the same `@InitiatingFlow` flow then on the receiving side you must be prepared to be initiated by any of the corresponding `initiateFlow()` calls! A good way of handling this ambiguity is to send as a first message a “role” message to the initiated flow, indicating which part of the initiating flow the rest of the counter-flow should conform to. For example send an enum, and on the other side start with a switch statement.

## SendAndReceive

We can also use a single call to send data to a counterparty and wait to receive data of a specific type back. The type of data sent doesn't need to match the type of the data received back:

```
val packet2: UntrustworthyData<Boolean> = counterpartySession.sendAndReceive<Boolean>(
    "You can send and receive any class!")
val boolean: Boolean = packet2.unwrap { data ->
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    data
}
```

```
UntrustworthyData<Boolean> packet2 = counterpartySession.sendAndReceive(Boolean.class,
    "You can send and receive any class!");
Boolean bool = packet2.unwrap(data -> {
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    return data;
});
```

## Counterparty response

Suppose we're now on the Responder side of the flow. We just received the following series of messages from the Initiator:

1. They sent us an Any instance
2. They waited to receive an Integer instance back
3. They sent a String instance and waited to receive a Boolean instance back

Our side of the flow must mirror these calls. We could do this as follows:

```
val any: Any = counterpartySession.receive<Any>().unwrap { data -> data }
val string: String = counterpartySession.sendAndReceive<String>(99).unwrap { data ->_
    data }
counterpartySession.send(true)
```

```
Object obj = counterpartySession.receive(Object.class).unwrap(data -> data);
String string = counterpartySession.sendAndReceive(String.class, 99).unwrap(data ->_
    data);
counterpartySession.send(true);
```

## Why sessions?

Before FlowSession s were introduced the send/receive API looked a bit different. They were functions on FlowLogic and took the address Party as argument. The platform internally maintained a mapping from Party to session, hiding sessions from the user completely.

Although this is a convenient API it introduces subtle issues where a message that was originally meant for a specific session may end up in another.

Consider the following contrived example using the old Party based API:

```
@InitiatingFlow
class LaunchSpaceshipFlow : FlowLogic<Unit>() {
    @Suspendable
```

(continues on next page)

(continued from previous page)

```

override fun call() {
    val shouldLaunchSpaceship = receive<Boolean>(getPresident()).unwrap { it }
    if (shouldLaunchSpaceship) {
        launchSpaceship()
    }
}

fun launchSpaceship() {

}

fun getPresident(): Party {
    TODO()
}
}

@InitiatedBy(LaunchSpaceshipFlow::class)
@InitiatingFlow
class PresidentSpaceshipFlow(val launcher: Party) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        val needCoffee = true
        send(getSecretary(), needCoffee)
        val shouldLaunchSpaceship = false
        send(launcher, shouldLaunchSpaceship)
    }

    fun getSecretary(): Party {
        TODO()
    }
}

@InitiatedBy(PresidentSpaceshipFlow::class)
class SecretaryFlow(val president: Party) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        // ignore
    }
}
}

```

```

@InitiatingFlow
class LaunchSpaceshipFlow extends FlowLogic<Void> {
    @Suspendable
    @Override
    public Void call() throws FlowException {
        boolean shouldLaunchSpaceship = receive(Boolean.class, getPresident()) .
        ↵unwrap(s -> s);
        if (shouldLaunchSpaceship) {
            launchSpaceship();
        }
        return null;
    }

    public void launchSpaceship() {

}

    public Party getPresident() {
}

```

(continues on next page)

(continued from previous page)

```

        throw new AbstractMethodError();
    }

}

@InitiatedBy(LaunchSpaceshipFlow.class)
@InitiatingFlow
class PresidentSpaceshipFlow extends FlowLogic<Void> {
    private final Party launcher;

    public PresidentSpaceshipFlow(Party launcher) {
        this.launcher = launcher;
    }

    @Suspendable
    @Override
    public Void call() {
        boolean needCoffee = true;
        send(getSecretary(), needCoffee);
        boolean shouldLaunchSpaceship = false;
        send(launcher, shouldLaunchSpaceship);
        return null;
    }

    public Party getSecretary() {
        throw new AbstractMethodError();
    }
}

@InitiatedBy(PresidentSpaceshipFlow.class)
class SecretaryFlow extends FlowLogic<Void> {
    private final Party president;

    public SecretaryFlow(Party president) {
        this.president = president;
    }

    @Suspendable
    @Override
    public Void call() {
        // ignore
        return null;
    }
}

```

The intention of the flows is very clear: LaunchSpaceshipFlow asks the president whether a spaceship should be launched. It is expecting a boolean reply. The president in return first tells the secretary that they need coffee, which is also communicated with a boolean. Afterwards the president replies to the launcher that they don't want to launch.

However the above can go horribly wrong when the launcher happens to be the same party getSecretary returns. In this case the boolean meant for the secretary will be received by the launcher!

This indicates that `Party` is not a good identifier for the communication sequence, and indeed the `Party` based API may introduce ways for an attacker to fish for information and even trigger unintended control flow like in the above case.

Hence we introduced `FlowSession`, which identifies the communication sequence. With `FlowSession`s the above set of flows would look like this:

```

@InitiatingFlow
class LaunchSpaceshipFlowCorrect : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        val presidentSession = initiateFlow(getPresident())
        val shouldLaunchSpaceship = presidentSession.receive<Boolean>().unwrap { it }
        if (shouldLaunchSpaceship) {
            launchSpaceship()
        }
    }

    fun launchSpaceship() {}

    fun getPresident(): Party {
        TODO()
    }
}

@InitiatedBy(LaunchSpaceshipFlowCorrect::class)
@InitiatingFlow
class PresidentSpaceshipFlowCorrect(val launcherSession: FlowSession) : FlowLogic
    <><Unit>() {
    @Suspendable
    override fun call() {
        val needCoffee = true
        val secretarySession = initiateFlow(getSecretary())
        secretarySession.send(needCoffee)
        val shouldLaunchSpaceship = false
        launcherSession.send(shouldLaunchSpaceship)
    }

    fun getSecretary(): Party {
        TODO()
    }
}

@InitiatedBy(PresidentSpaceshipFlowCorrect::class)
class SecretaryFlowCorrect(val presidentSession: FlowSession) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        // ignore
    }
}

```

```

@InitiatingFlow
class LaunchSpaceshipFlowCorrect extends FlowLogic<Void> {
    @Suspendable
    @Override
    public Void call() throws FlowException {
        FlowSession presidentSession = initiateFlow(getPresident());
        boolean shouldLaunchSpaceship = presidentSession.receive(Boolean.class).
        <>unwrap(s -> s);
        if (shouldLaunchSpaceship) {
            launchSpaceship();
        }
        return null;
    }
}

```

(continues on next page)

(continued from previous page)

```
}

public void launchSpaceship() {
}

public Party getPresident() {
    throw new AbstractMethodError();
}
}

@InitiatedBy(LaunchSpaceshipFlowCorrect.class)
@InitiatingFlow
class PresidentSpaceshipFlowCorrect extends FlowLogic<Void> {
    private final FlowSession launcherSession;

    public PresidentSpaceshipFlowCorrect(FlowSession launcherSession) {
        this.launcherSession = launcherSession;
    }

    @Suspendable
    @Override
    public Void call() {
        boolean needCoffee = true;
        FlowSession secretarySession = initiateFlow(getSecretary());
        secretarySession.send(needCoffee);
        boolean shouldLaunchSpaceship = false;
        launcherSession.send(shouldLaunchSpaceship);
        return null;
    }

    public Party getSecretary() {
        throw new AbstractMethodError();
    }
}

@InitiatedBy(PresidentSpaceshipFlowCorrect.class)
class SecretaryFlowCorrect extends FlowLogic<Void> {
    private final FlowSession presidentSession;

    public SecretaryFlowCorrect(FlowSession presidentSession) {
        this.presidentSession = presidentSession;
    }

    @Suspendable
    @Override
    public Void call() {
        // ignore
        return null;
    }
}
```

Note how the president is now explicit about which session it wants to send to.

## Porting from the old Party-based API

In the old API the first send or receive to a Party was the one kicking off the counter-flow. This is now explicit in the `initiateFlow` function call. To port existing code:

```
send(regulator, Any()) // Old API
// becomes
val session = initiateFlow(regulator)
session.send(Any())
```

```
send(regulator, new Object()); // Old API
// becomes
FlowSession session = initiateFlow(regulator);
session.send(new Object());
```

## 4.7.7 Subflows

Subflows are pieces of reusable flows that may be run by calling `FlowLogic.subFlow`. There are two broad categories of subflows, inlined and initiating ones. The main difference lies in the counter-flow's starting method, initiating ones initiate counter-flows automatically, while inlined ones expect some parent counter-flow to run the inlined counterpart.

### Inlined subflows

Inlined subflows inherit their calling flow's type when initiating a new session with a counterparty. For example, say we have flow A calling an inlined subflow B, which in turn initiates a session with a party. The `FlowLogic` type used to determine which counter-flow should be kicked off will be A, not B. Note that this means that the other side of this inlined flow must therefore be implemented explicitly in the kicked off flow as well. This may be done by calling a matching inlined counter-flow, or by implementing the other side explicitly in the kicked off parent flow.

An example of such a flow is `CollectSignaturesFlow`. It has a counter-flow `SignTransactionFlow` that isn't annotated with `InitiatedBy`. This is because both of these flows are inlined; the kick-off relationship will be defined by the parent flows calling `CollectSignaturesFlow` and `SignTransactionFlow`.

In the code inlined subflows appear as regular `FlowLogic` instances, *without* either of the `@InitiatingFlow` or `@InitiatedBy` annotation.

---

**Note:** Inlined flows aren't versioned; they inherit their parent flow's version.

---

### Initiating subflows

Initiating subflows are ones annotated with the `@InitiatingFlow` annotation. When such a flow initiates a session its type will be used to determine which `@InitiatedBy` flow to kick off on the counterparty.

An example is the `@InitiatingFlow InitiatorFlow/@InitiatedBy ResponderFlow` flow pair in the `FlowCookbook`.

---

**Note:** Initiating flows are versioned separately from their parents.

---

---

**Note:** The only exception to this rule is `FinalityFlow` which is annotated with `@InitiatingFlow` but is an inlined flow. This flow was previously initiating and the annotation exists to maintain backwards compatibility with old code.

---

### Core initiating subflows

Corda-provided initiating subflows are a little different to standard ones as they are versioned together with the platform, and their initiated counter-flows are registered explicitly, so there is no need for the `InitiatedBy` annotation.

### Library flows

Corda installs four initiating subflow pairs on each node by default:

- `NotaryChangeFlow/NotaryChangeHandler`, which should be used to change a state's notary
- `ContractUpgradeFlow.Initiate/ContractUpgradeHandler`, which should be used to change a state's contract
- `SwapIdentitiesFlow/SwapIdentitiesHandler`, which is used to exchange confidential identities with a counterparty

**Warning:** `SwapIdentitiesFlow/SwapIdentitiesHandler` are only installed if the confidential-identities module is included. The confidential-identities module is still not stabilised, so the `SwapIdentitiesFlow/SwapIdentitiesHandler` API may change in future releases. See [Corda API](#).

Corda also provides a number of built-in inlined subflows that should be used for handling common tasks. The most important are:

- `FinalityFlow` which is used to notarise, record locally and then broadcast a signed transaction to its participants and any extra parties.
- `ReceiveFinalityFlow` to receive these notarised transactions from the `FinalityFlow` sender and record locally.
- `CollectSignaturesFlow`, which should be used to collect a transaction's required signatures
- `SendTransactionFlow`, which should be used to send a signed transaction if it needed to be resolved on the other side.
- `ReceiveTransactionFlow`, which should be used receive a signed transaction

Let's look at some of these flows in more detail.

### FinalityFlow

`FinalityFlow` allows us to notarise the transaction and get it recorded in the vault of the participants of all the transaction's states:

```
val notarisedTx1: SignedTransaction = subFlow(FinalityFlow(fullySignedTx,  
    listOf(counterpartySession), FINALISATION.childProgressTracker()))
```

```
SignedTransaction notarisedTx1 = subFlow(new FinalityFlow(fullySignedTx,_
    ↪singleton(counterpartySession), FINALISATION.childProgressTracker()));
```

We can also choose to send the transaction to additional parties who aren't one of the state's participants:

```
val partySessions: List<FlowSession> = listOf(counterpartySession,_
    ↪initiateFlow(regulator))
val notarisedTx2: SignedTransaction = subFlow(FinalityFlow(fullySignedTx,_
    ↪partySessions, FINALISATION.childProgressTracker()))
```

```
List<FlowSession> partySessions = Arrays.asList(counterpartySession,_
    ↪initiateFlow(regulator));
SignedTransaction notarisedTx2 = subFlow(new FinalityFlow(fullySignedTx,_
    ↪partySessions, FINALISATION.childProgressTracker()));
```

Only one party has to call `FinalityFlow` for a given transaction to be recorded by all participants. It **must not** be called by every participant. Instead, every other participant **must** call `ReceiveFinalityFlow` in their responder flow to receive the transaction:

```
subFlow(ReceiveFinalityFlow(counterpartySession, expectedTxId = idOfTxWeSigned))
```

```
subFlow(new ReceiveFinalityFlow(counterpartySession, idOfTxWeSigned));
```

`idOfTxWeSigned` is an optional parameter used to confirm that we got the right transaction. It comes from using `SignTransactionFlow` which is described below.

### Error handling behaviour

Once a transaction has been notarised and its input states consumed by the flow initiator (eg. sender), should the participant(s) receiving the transaction fail to verify it, or the receiving flow (the finality handler) fails due to some other error, we then have a scenario where not all parties have the correct up to date view of the ledger (a condition where eventual consistency between participants takes longer than is normally the case under Corda's [eventual consistency model](#)). To recover from this scenario, the receiver's finality handler will automatically be sent to the node-flow-hospital where it's suspended and retried from its last checkpoint upon node restart, or according to other conditional retry rules explained in flow hospital runtime behaviour. This gives the node operator the opportunity to recover from the error. Until the issue is resolved the node will continue to retry the flow on each startup. Upon successful completion by the receiver's finality flow, the ledger will become fully consistent once again.

**Warning:** It's possible to forcibly terminate the erroring finality handler using the `killFlow` RPC but at the risk of an inconsistent view of the ledger.

---

**Note:** A future release will allow retrying hospitalised flows without restarting the node, i.e. via RPC.

---

### CollectSignaturesFlow/SignTransactionFlow

The list of parties who need to sign a transaction is dictated by the transaction's commands. Once we've signed a transaction ourselves, we can automatically gather the signatures of the other required signers using `CollectSignaturesFlow`:

```
val fullySignedTx: SignedTransaction = subFlow(CollectSignaturesFlow(twiceSignedTx,_
    ↪setOf(counterpartySession, regulatorSession), SIGS_GATHERING,
    ↪childProgressTracker()))
```

(continues on next page)

(continued from previous page)

```
SignedTransaction fullySignedTx = subFlow(new CollectSignaturesFlow(twiceSignedTx,
    ↪emptySet(), SIGS_GATHERING.childProgressTracker()));
```

Each required signer will need to respond by invoking its own `SignTransactionFlow` subclass to check the transaction (by implementing the `checkTransaction` method) and provide their signature if they are satisfied:

```
val signTransactionFlow: SignTransactionFlow = object :_
    ↪SignTransactionFlow(counterpartySession) {
    override fun checkTransaction(stx: SignedTransaction) = requireThat {
        // Any additional checking we see fit...
        val outputState = stx.tx.outputsOfType<DummyState>().single()
        require(outputState.magicNumber == 777)
    }
}

val idOfTxWeSigned = subFlow(signTransactionFlow).id
```

```
class SignTxFlow extends SignTransactionFlow {
    private SignTxFlow(FlowSession otherSession, ProgressTracker progressTracker) {
        super(otherSession, progressTracker);
    }

    @Override
    protected void checkTransaction(SignedTransaction stx) {
        requireThat(require -> {
            // Any additional checking we see fit...
            DummyState outputState = (DummyState) stx.getTx().getOutputs().get(0).
                ↪getData();
            checkArgument(outputState.getMagicNumber() == 777);
            return null;
        });
    }
}

SecureHash idOfTxWeSigned = subFlow(new SignTxFlow(counterpartySession,
    ↪SignTransactionFlow.tracker())).getId();
```

Types of things to check include:

- Ensuring that the transaction received is the expected type, i.e. has the expected type of inputs and outputs
- Checking that the properties of the outputs are expected, this is in the absence of integrating reference data sources to facilitate this
- Checking that the transaction is not incorrectly spending (perhaps maliciously) asset states, as potentially the transaction creator has access to some of signer's state references

## SendTransactionFlow/ReceiveTransactionFlow

Verifying a transaction received from a counterparty also requires verification of every transaction in its dependency chain. This means the receiving party needs to be able to ask the sender all the details of the chain. The sender will use `SendTransactionFlow` for sending the transaction and then for processing all subsequent transaction data vending requests as the receiver walks the dependency chain using `ReceiveTransactionFlow`:

```
subFlow(SendTransactionFlow(counterpartySession, twiceSignedTx))

// Optional request verification to further restrict data access.
subFlow(object : SendTransactionFlow(counterpartySession, twiceSignedTx) {
    override fun verifyDataRequest(dataRequest: FetchDataFlow.Request.Data) {
        // Extra request verification.
    }
})
```

```
subFlow(new SendTransactionFlow(counterpartySession, twiceSignedTx));

// Optional request verification to further restrict data access.
subFlow(new SendTransactionFlow(counterpartySession, twiceSignedTx) {
    @Override
    protected void verifyDataRequest(@NotNull FetchDataFlow.Request.Data dataRequest)
    {
        // Extra request verification.
    }
});
```

We can receive the transaction using `ReceiveTransactionFlow`, which will automatically download all the dependencies and verify the transaction:

```
val verifiedTransaction = subFlow(ReceiveTransactionFlow(counterpartySession))
```

```
SignedTransaction verifiedTransaction = subFlow(new_
->ReceiveTransactionFlow(counterpartySession));
```

We can also send and receive a `StateAndRef` dependency chain and automatically resolve its dependencies:

```
subFlow(SendStateAndRefFlow(counterpartySession, dummyStates))

// On the receive side ...
val resolvedStateAndRef = subFlow(ReceiveStateAndRefFlow<DummyState>
->(counterpartySession))
```

```
subFlow(new SendStateAndRefFlow(counterpartySession, dummyStates));

// On the receive side ...
List<StateAndRef<DummyState>> resolvedStateAndRef = subFlow(new ReceiveStateAndRefFlow
-><>(counterpartySession));
```

## Why inlined subflows?

Inlined subflows provide a way to share commonly used flow code *while forcing users to create a parent flow*. Take for example `CollectSignaturesFlow`. Say we made it an initiating flow that automatically kicks off `SignTransactionFlow` that signs the transaction. This would mean malicious nodes can just send any old transaction to us using `CollectSignaturesFlow` and we would automatically sign it!

By making this pair of flows inlined we provide control to the user over whether to sign the transaction or not by forcing them to nest it in their own parent flows.

In general if you're writing a subflow the decision of whether you should make it initiating should depend on whether the counter-flow needs broader context to achieve its goal.

## 4.7.8 FlowException

Suppose a node throws an exception while running a flow. Any counterparty flows waiting for a message from the node (i.e. as part of a call to `receive` or `sendAndReceive`) will be notified that the flow has unexpectedly ended and will themselves end. However, the exception thrown will not be propagated back to the counterparties.

If you wish to notify any waiting counterparties of the cause of the exception, you can do so by throwing a `FlowException`:

```
/*
 * Exception which can be thrown by a [FlowLogic] at any point in its logic to
 * unexpectedly bring it to a permanent end.
 * The exception will propagate to all counterparty flows and will be thrown on their
 * end the next time they wait on a
 * [FlowSession.receive] or [FlowSession.sendAndReceive]. Any flow which no longer
 * needs to do a receive, or has already
 * ended, will not receive the exception (if this is required then have them wait for
 * a confirmation message).
 *
 * If the *rethrown* [FlowException] is uncaught in counterparty flows and
 * propagation triggers then the exception is
 * downgraded to an [UnexpectedFlowEndException]. This means only immediate
 * counterparty flows will receive information
 * about what the exception was.
 *
 * [FlowException] (or a subclass) can be a valid expected response from a flow,
 * particularly ones which act as a service.
 * It is recommended a [FlowLogic] document the [FlowException] types it can throw.
 *
 * @property originalErrorId the ID backing [getErrorHandler]. If null it will be set
 * dynamically by the flow framework when
 * the exception is handled. This ID is propagated to counterparty flows, even
 * when the [FlowException] is
 * downgraded to an [UnexpectedFlowEndException]. This is so the error conditions
 * may be correlated later on.
 */
open class FlowException(message: String?, cause: Throwable?, var originalErrorId: Long? = null) :
    CordaException(message, cause), IdentifiableException {
    constructor(message: String?, cause: Throwable?) : this(message, cause, null)
    constructor(message: String?) : this(message, null)
    constructor(cause: Throwable?) : this(cause?.toString(), cause)
    constructor() : this(null, null)

    // private field with obscure name to ensure it is not overridden
    private var peer: Party? = null

    override fun getErrorHandler(): Long? = originalErrorId
}
```

The flow framework will automatically propagate the `FlowException` back to the waiting counterparties.

There are many scenarios in which throwing a `FlowException` would be appropriate:

- A transaction doesn't verify()
- A transaction's signatures are invalid
- The transaction does not match the parameters of the deal as discussed

- You are renegeing on a deal

### 4.7.9 ProgressTracker

We can give our flow a progress tracker. This allows us to see the flow's progress visually in our node's CRaSH shell.

To provide a progress tracker, we have to override `FlowLogic.progressTracker` in our flow:

```
companion object {
    object ID_OTHER_NODES : Step("Identifying other nodes on the network.")
    object SENDING_AND RECEIVING_DATA : Step("Sending data between parties.")
    object EXTRACTING_VAULT_STATES : Step("Extracting states from the vault.")
    object OTHER_TX_COMPONENTS : Step("Gathering a transaction's other components.")
    object TX_BUILDING : Step("Building a transaction.")
    object TX_SIGNING : Step("Signing a transaction.")
    object TX_VERIFICATION : Step("Verifying a transaction.")
    object SIGS_GATHERING : Step("Gathering a transaction's signatures.") {
        // Wiring up a child progress tracker allows us to see the
        // subflow's progress steps in our flow's progress tracker.
        override fun childProgressTracker() = CollectSignaturesFlow.tracker()
    }

    object VERIFYING_SIGS : Step("Verifying a transaction's signatures.")
    object FINALISATION : Step("Finalising a transaction.") {
        override fun childProgressTracker() = FinalityFlow.tracker()
    }
}

fun tracker() = ProgressTracker(
    ID_OTHER_NODES,
    SENDING_AND RECEIVING_DATA,
    EXTRACTING_VAULT_STATES,
    OTHER_TX_COMPONENTS,
    TX_BUILDING,
    TX_SIGNING,
    TX_VERIFICATION,
    SIGS_GATHERING,
    VERIFYING_SIGS,
    FINALISATION
)
}
```

```
private static final Step ID_OTHER_NODES = new Step("Identifying other nodes on the_
network.");
private static final Step SENDING_AND RECEIVING_DATA = new Step("Sending data between_
parties.");
private static final Step EXTRACTING_VAULT_STATES = new Step("Extracting states from_
the vault.");
private static final Step OTHER_TX_COMPONENTS = new Step("Gathering a transaction's_
other components.");
private static final Step TX_BUILDING = new Step("Building a transaction.");
private static final Step TX_SIGNING = new Step("Signing a transaction.");
private static final Step TX_VERIFICATION = new Step("Verifying a transaction.");
private static final Step SIGS_GATHERING = new Step("Gathering a transaction's_
signatures.") {
    // Wiring up a child progress tracker allows us to see the
    // subflow's progress steps in our flow's progress tracker.
    @Override
```

(continues on next page)

(continued from previous page)

```
public ProgressTracker childProgressTracker() {
    return CollectSignaturesFlow.tracker();
}

};

private static final Step VERIFYING_SIGS = new Step("Verifying a transaction's
→signatures.");
private static final Step FINALISATION = new Step("Finalising a transaction.") {
    @Override
    public ProgressTracker childProgressTracker() {
        return FinalityFlow.tracker();
    }
};

private final ProgressTracker progressTracker = new ProgressTracker(
    ID_OTHER_NODES,
    SENDING_AND RECEIVING_DATA,
    EXTRACTING_VAULT_STATES,
    OTHER_TX_COMPONENTS,
    TX_BUILDING,
    TX_SIGNING,
    TX_VERIFICATION,
    SIGS_GATHERING,
    FINALISATION
);
```

We then update the progress tracker's current step as we progress through the flow as follows:

```
progressTracker.currentStep = ID_OTHER_NODES
```

```
progressTracker.setCurrentStep(ID_OTHER_NODES);
```

#### 4.7.10 HTTP and database calls

HTTP, database and other calls to external resources are allowed in flows. However, their support is currently limited:

- The call must be executed in a BLOCKING way. Flows don't currently support suspending to await the response to a call to an external resource
  - For this reason, the call should be provided with a timeout to prevent the flow from suspending forever. If the timeout elapses, this should be treated as a soft failure and handled by the flow's business logic
- The call must be idempotent. If the flow fails and has to restart from a checkpoint, the call will also be replayed

#### 4.7.11 Concurrency, Locking and Waiting

Corda is designed to:

- run many flows in parallel
- persist flows to storage and resurrect those flows much later
- (in the future) migrate flows between JVMs

Because of this, care must be taken when performing locking or waiting operations.

## Locking

Flows should avoid using locks or interacting with objects that are shared between flows (except for ServiceHub and other carefully crafted services such as Oracles. See [Writing oracle services](#)). Locks will significantly reduce the scalability of the node, and can cause the node to deadlock if they remain locked across flow context switch boundaries (such as when sending and receiving from peers, as discussed above, or sleeping, as discussed below).

## Waiting

A flow can wait until a specific transaction has been received and verified by the node using `FlowLogic.waitForLedgerCommit`. Outside of this, scheduling an activity to occur at some future time should be achieved using `SchedulableState`.

However, if there is a need for brief pauses in flows, you have the option of using `FlowLogic.sleep` in place of where you might have used `Thread.sleep`. Flows should expressly not use `Thread.sleep`, since this will prevent the node from processing other flows in the meantime, significantly impairing the performance of the node.

Even `FlowLogic.sleep` should not be used to create long running flows or as a substitute to using the `SchedulableState` scheduler, since the Corda ethos is for short-lived flows (long-lived flows make upgrading nodes or CorDapps much more complicated).

For example, the finance package currently uses `FlowLogic.sleep` to make several attempts at coin selection when many states are soft locked, to wait for states to become unlocked:

```
for (retryCount in 1..maxRetries) {
    if (!attemptSpend(services, amount, lockId, notary,
        ↪onlyFromIssuerParties, withIssuerRefs, stateAndRefs)) {
        log.warn("Coin selection failed on attempt $retryCount")
        // TODO: revisit the back off strategy for contended spending.
        if (retryCount != maxRetries) {
            stateAndRefs.clear()
            val durationMillis = (minOf(retrySleep.shl(retryCount), retryCap,
                ↪/ 2) * (1.0 + Math.random())).toInt()
            FlowLogic.sleep(durationMillis.millis)
        } else {
            log.warn("Insufficient spendable states identified for $amount")
        }
    } else {
        break
    }
}
```

## 4.8 API: Identity

### Contents

- [API: Identity](#)
  - [Party](#)
  - [Confidential identities](#)
    - \* [SwapIdentitiesFlow](#)
    - \* [IdentitySyncFlow](#)

## 4.8.1 Party

Parties on the network are represented using the `AbstractParty` class. There are two types of `AbstractParty`:

- `Party`, identified by a `PublicKey` and a `CordaX500Name`
- `AnonymousParty`, identified by a `PublicKey` only

Using `AnonymousParty` to identify parties in states and commands prevents nodes from learning the identities of the parties involved in a transaction when they verify the transaction's dependency chain. When preserving the anonymity of each party is not required (e.g. for internal processing), `Party` can be used instead.

The identity service allows flows to resolve `AnonymousParty` to `Party`, but only if the anonymous party's identity has already been registered with the node (typically handled by `SwapIdentitiesFlow` or `IdentitySyncFlow`, discussed below).

`Party` names use the `CordaX500Name` data class, which enforces the structure of names within Corda, as well as ensuring a consistent rendering of the names in plain text.

Support for both `Party` and `AnonymousParty` classes in Corda enables sophisticated selective disclosure of identity information. For example, it is possible to construct a transaction using an `AnonymousParty` (so nobody can learn of your involvement by inspection of the transaction), yet prove to specific counterparties that this `AnonymousParty` actually corresponds to your well-known identity. This is achieved using the `PartyAndCertificate` data class, which contains the X.509 certificate path proving that a given `AnonymousParty` corresponds to a given `Party`. Each `PartyAndCertificate` can be propagated to counterparties on a need-to-know basis.

The `PartyAndCertificate` class is also used by the network map service to represent well-known identities, with the certificate path proving the certificate was issued by the doorman service.

## 4.8.2 Confidential identities

**Warning:** The `confidential-identities` module is still not stabilised, so this API may change in future releases. See [Corda API](#).

Confidential identities are key pairs where the corresponding X.509 certificate (and path) are not made public, so that parties who are not involved in the transaction cannot identify the owner. They are owned by a well-known identity, which must sign the X.509 certificate. Before constructing a new transaction the involved parties must generate and exchange new confidential identities, a process which is managed using `SwapIdentitiesFlow` (discussed below). The public keys of these confidential identities are then used when generating output states and commands for the transaction.

Where using outputs from a previous transaction in a new transaction, counterparties may need to know who the involved parties are. One example is the `TwoPartyTradeFlow`, where an existing asset is exchanged for cash. If confidential identities are being used, the buyer will want to ensure that the asset being transferred is owned by the seller, and the seller will likewise want to ensure that the cash being transferred is owned by the buyer. Verifying this requires both nodes to have a copy of the confidential identities for the asset and cash input states. `IdentitySyncFlow` manages this process. It takes as inputs a transaction and a counterparty, and for every confidential identity involved in that transaction for which the calling node holds the certificate path, it sends this certificate path to the counterparty.

### SwapIdentitiesFlow

`SwapIdentitiesFlow` is typically run as a subflow of another flow. It takes as its sole constructor argument the counterparty we want to exchange confidential identities with. It returns a mapping from the identities of the caller

and the counterparty to their new confidential identities. In the future, this flow will be extended to handle swapping identities with multiple parties at once.

You can see an example of using `SwapIdentitiesFlow` in `TwoPartyDealFlow.kt`:

```
@Suspendable
override fun call(): SignedTransaction {
    progressTracker.currentStep = GENERATING_ID
    val txIdentities = subFlow(SwapIdentitiesFlow(otherSideSession))
    val anonymousMe = txIdentities[ourIdentity]!!
    val anonymousCounterparty = txIdentities[otherSideSession.counterparty]!!
```

`SwapIdentitiesFlow` goes through the following key steps:

1. Generate a new confidential identity from our well-known identity
2. Create a `CertificateOwnershipAssertion` object containing the new confidential identity (X500 name, public key)
3. Sign this object with the confidential identity's private key
4. Send the confidential identity and aforementioned signature to counterparties, while receiving theirs
5. Verify the signatures to ensure that identities were generated by the involved set of parties
6. Verify the confidential identities are owned by the expected well known identities
7. Store the confidential identities and return them to the calling flow

This ensures not only that the confidential identity X.509 certificates are signed by the correct well-known identities, but also that the confidential identity private key is held by the counterparty, and that a party cannot claim ownership of another party's confidential identities.

## IdentitySyncFlow

When constructing a transaction whose input states reference confidential identities, it is common for counterparties to require knowledge of which well-known identity each confidential identity maps to. `IdentitySyncFlow` handles this process. You can see an example of its use in `TwoPartyTradeFlow.kt`.

`IdentitySyncFlow` is divided into two parts:

- `IdentitySyncFlow.Send`
- `IdentitySyncFlow.Receive`

`IdentitySyncFlow.Send` is invoked by the party initiating the identity synchronization:

```
// 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
```

The identity synchronization flow goes through the following key steps:

1. Extract participant identities from all input and output states and remove any well known identities. Required signers on commands are currently ignored as they are presumed to be included in the participants on states, or to be well-known identities of services (such as an oracle service)
2. For each counterparty node, send a list of the public keys of the confidential identities, and receive back a list of those the counterparty needs the certificate path for
3. Verify the requested list of identities contains only confidential identities in the offered list, and abort otherwise
4. Send the requested confidential identities as `PartyAndCertificate` instances to the counterparty

---

**Note:** `IdentitySyncFlow` works on a push basis. The initiating node can only send confidential identities it has the X.509 certificates for, and the remote nodes can only request confidential identities being offered (are referenced in the transaction passed to the initiating flow). There is no standard flow for nodes to collect confidential identities before assembling a transaction, and this is left for individual flows to manage if required.

---

Meanwhile, `IdentitySyncFlow.Receive` is invoked by all the other (non-initiating) parties involved in the identity synchronization process:

```
// 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))
```

`IdentitySyncFlow` will serve all confidential identities in the provided transaction, irrespective of well-known identity. This is important for more complex transaction cases with 3+ parties, for example:

- Alice is building the transaction, and provides some input state *x* owned by a confidential identity of Alice
- Bob provides some input state *y* owned by a confidential identity of Bob
- Charlie provides some input state *z* owned by a confidential identity of Charlie

Alice may know all of the confidential identities ahead of time, but Bob not know about Charlie's and vice-versa. The assembled transaction therefore has three input states *x*, *y* and *z*, for which only Alice possesses certificates for all confidential identities. `IdentitySyncFlow` must send not just Alice's confidential identity but also any other identities in the transaction to the Bob and Charlie.

## 4.9 API: ServiceHub

Within `FlowLogic.call`, the flow developer has access to the node's `ServiceHub`, which provides access to the various services the node provides. The services offered by the `ServiceHub` are split into the following categories:

- **`ServiceHub.networkMapCache`**
  - Provides information on other nodes on the network (e.g. notaries...)
- **`ServiceHub.identityService`**
  - Allows you to resolve anonymous identities to well-known identities if you have the required certificates
- **`ServiceHub.attachments`**
  - Gives you access to the node's attachments
- **`ServiceHub.validatedTransactions`**
  - Gives you access to the transactions stored in the node

- **ServiceHub.vaultService**
  - Stores the node's current and historic states
- **ServiceHub.keyManagementService**
  - Manages signing transactions and generating fresh public keys
- **ServiceHub.myInfo**
  - Other information about the node
- **ServiceHub.clock**
  - Provides access to the node's internal time and date

Additional, ServiceHub exposes the following properties:

- ServiceHub.loadState and ServiceHub.toStateAndRef to resolve a StateRef into a TransactionState or a StateAndRef
- ServiceHub.signInitialTransaction to sign a TransactionBuilder and convert it into a SignedTransaction
- ServiceHub.createSignature and ServiceHub.addSignature to create and add signatures to a SignedTransaction

## 4.10 API: Service Classes

Service classes are long-lived instances that can trigger or be triggered by flows from within a node. A Service class is limited to a single instance per node. During startup, the node handles the creation of the service.

Services allow related, reusable, functions to be separated into their own class where their functionality is grouped together. These functions can then be called from other services or flows.

### 4.10.1 Creating a Service

To define a Service class:

- Add the CordaService annotation
- Add a constructor with a single parameter of AppServiceHub
- Extend SingletonSerializeAsToken

Below is an empty implementation of a Service class:

```
@CordaService
class MyCordaService(private val serviceHub: AppServiceHub) : SingletonSerializeAsToken() {

    init {
        // code ran at service creation / node startup
    }

    // public api of service
}
```

```
@CordaService
public class MyCordaService extends SingletonSerializeAsToken {

    private AppServiceHub serviceHub;

    public MyCordaService(AppServiceHub serviceHub) {
        this.serviceHub = serviceHub;
        // code ran at service creation / node startup
    }

    // public api of service
}
```

The AppServiceHub provides the ServiceHub functionality to the Service class, with the extra ability to start flows. Starting flows from AppServiceHub is explained further in [Starting Flows from a Service](#).

Code can be run during node startup when the class is being initialised. To do so, place the code into the `init` block or constructor. This is useful when a service needs to establish a connection to an external database or setup observables via `ServiceHub.trackBy` during its startup. These can then persist during the service's lifetime.

#### 4.10.2 Retrieving a Service

A Service class can be retrieved by calling `ServiceHub.cordaService` which returns the single instance of the class passed into the function:

```
val service: MyCordaService = serviceHub.cordaService(MyCordaService::class.java)
```

```
MyCordaService service = serviceHub.cordaService(MyCordaService.class);
```

**Warning:** `ServiceHub.cordaService` should not be called during initialisation of a flow and should instead be called in line where needed or set after the flow's `call` function has been triggered.

#### 4.10.3 Starting Flows from a Service

Starting flows via a service can lead to deadlock within the node's flow worker queue, which will prevent new flows from starting. To avoid this, the rules below should be followed:

- When called from a running flow, the service must invoke the new flow from another thread. The existing flow cannot await the execution of the new flow.
- When `ServiceHub.trackBy` is placed inside the service, flows started inside the observable must be placed onto another thread.
- Flows started by other means, do not require any special treatment.

---

**Note:** It is possible to avoid deadlock without following these rules depending on the number of flows running within the node. But, if the number of flows violating these rules reaches the flow worker queue size, then the node will deadlock. It is best practice to abide by these rules to remove this possibility.

---

## 4.11 API: RPC operations

The node's owner interacts with the node solely via remote procedure calls (RPC). The node's owner does not have access to the node's ServiceHub.

The key RPC operations exposed by the node are:

- **CordaRPCOps.vaultQueryBy**
  - Extract states from the node's vault based on a query criteria
- **CordaRPCOps.vaultTrackBy**
  - As above, but also returns an observable of future states matching the query
- **CordaRPCOps.networkMapFeed**
  - A list of network nodes, and an observable of changes to the network map
- **CordaRPCOps.registeredFlows**
  - See a list of registered flows on the node
- **CordaRPCOps.startFlowDynamic**
  - Start one of the node's registered flows
- **CordaRPCOps.startTrackedFlowDynamic**
  - As above, but also returns a progress handle for the flow
- **CordaRPCOps.nodeInfo**
  - Returns information about the node
- **CordaRPCOps.currentTime**
  - Returns the current time according to the node's clock
- **CordaRPCOps.partyFromKey/CordaRPCOps.wellKnownPartyFromX500Name**
  - Retrieves a party on the network based on a public key or X500 name
- **CordaRPCOps.uploadAttachment/CordaRPCOps.openAttachment/CordaRPCOps.attachmentExists**
  - Uploads, opens and checks for the existence of attachments

## 4.12 API: Core types

### Contents

- *API: Core types*
  - *SecureHash*
  - *CompositeKey*

Corda provides several more core classes as part of its API.

### 4.12.1 SecureHash

The `SecureHash` class is used to uniquely identify objects such as transactions and attachments by their hash. Any object that needs to be identified by its hash should implement the `NamedByHash` interface:

```
/** Implemented by anything that can be named by a secure hash value (e.g. transactions, attachments). */
interface NamedByHash {
    val id: SecureHash
}
```

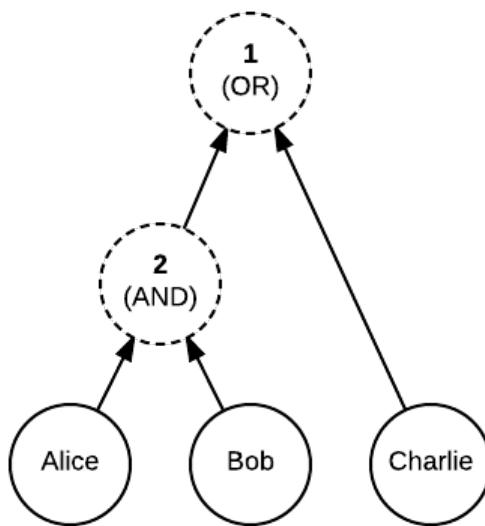
`SecureHash` is a sealed class that only defines a single subclass, `SecureHash.SHA256`. There are utility methods to create and parse `SecureHash.SHA256` objects.

### 4.12.2 CompositeKey

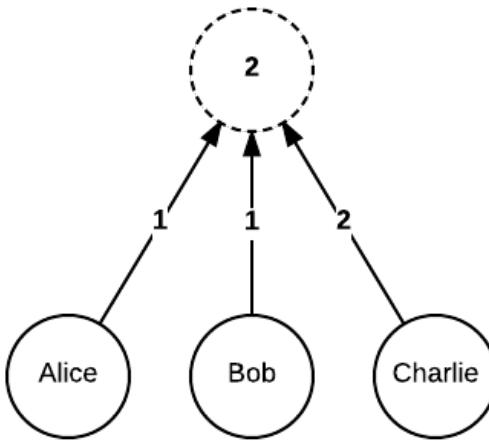
Corda supports scenarios where more than one signature is required to authorise a state object transition. For example: “Either the CEO or 3 out of 5 of his assistants need to provide signatures”.

This is achieved using a `CompositeKey`, which uses public-key composition to organise the various public keys into a tree data structure. A `CompositeKey` is a tree that stores the cryptographic public key primitives in its leaves and the composition logic in the intermediary nodes. Every intermediary node specifies a *threshold* of how many child signatures it requires.

An illustration of an “either Alice and Bob, or Charlie” composite key:



To allow further flexibility, each child node can have an associated custom *weight* (the default is 1). The *threshold* then specifies the minimum total weight of all children required. Our previous example can also be expressed as:



Signature verification is performed in two stages:

1. Given a list of signatures, each signature is verified against the expected content.
2. The public keys corresponding to the signatures are matched against the leaves of the composite key tree in question, and the total combined weight of all children is calculated for every intermediary node. If all thresholds are satisfied, the composite key requirement is considered to be met.

## 4.13 API: Testing

### Contents

- *API: Testing*
  - *Flow testing*
    - \* *MockNetwork*
    - \* *Adding nodes to the network*
    - \* *Running the network*
    - \* *Running flows*
    - \* *Accessing StartedMockNode internals*
      - *Querying a node's vault*
      - *Examining a node's transaction storage*
    - \* *Further examples*
  - *Contract testing*
    - \* *Test identities*
    - \* *MockServices*
    - \* *Writing tests using a test ledger*
      - *Checking for failure states*

- *Testing multiple scenarios at once*
- *Chaining transactions*
- \* *Further examples*

### 4.13.1 Flow testing

#### MockNetwork

Flow testing can be fully automated using a MockNetwork composed of StartedMockNode nodes. Each StartedMockNode behaves like a regular Corda node, but its services are either in-memory or mocked out.

A MockNetwork is created as follows:

```
import net.corda.core.identity.CordaX500Name
import net.corda.testing.node.MockNetwork
import net.corda.testing.node.MockNetworkParameters
import net.corda.testing.node.StartedMockNode
import net.corda.testing.node.TestCordapp.Companion.findCordapp
import org.junit.After
import org.junit.Before

class MockNetworkTestsTutorial {

    private val mockNet = MockNetwork(MockNetworkParameters(listOf(findCordapp("com.
    ↪mycordapp.package"))))

    @After
    fun cleanUp() {
        mockNet.stopNodes()
    }
}
```

```
import net.corda.core.identity.CordaX500Name;
import net.corda.testing.node.MockNetwork;
import net.corda.testing.node.MockNetworkParameters;
import net.corda.testing.node.StartedMockNode;
import org.junit.After;
import org.junit.Before;

import static java.util.Collections.singletonList;
import static net.corda.testing.node.TestCordapp.findCordapp;

public class MockNetworkTestsTutorial {

    private final MockNetwork mockNet = new MockNetwork(new_
    ↪MockNetworkParameters(singletonList(findCordapp("com.mycordapp.package"))));

    @After
    public void cleanUp() {
        mockNet.stopNodes();
    }
}
```

The MockNetwork requires at a minimum a list of CorDapps to be installed on each StartedMockNode. The CorDapps are looked up on the classpath by package name, using TestCordapp.findCordapp. TestCordapp.findCordapp scans the current classpath to find the CorDapp that contains the given package. This includes all the

associated CorDapp metadata present in its MANIFEST.

`MockNetworkParameters` provides other properties for the network which can be tweaked. They default to sensible values if not specified.

## Adding nodes to the network

Nodes are created on the `MockNetwork` using:

```
private lateinit var nodeA: StartedMockNode
private lateinit var nodeB: StartedMockNode

@Before
fun setUp() {
    nodeA = mockNet.createNode()
    // We can optionally give the node a name.
    nodeB = mockNet.createNode(CordaX500Name("Bank B", "London", "GB"))
}
```

```
private StartedMockNode nodeA;
private StartedMockNode nodeB;

@Before
public void setUp() {
    nodeA = mockNet.createNode();
    // We can optionally give the node a name.
    nodeB = mockNet.createNode(new CordaX500Name("Bank B", "London", "GB"));
}
```

Nodes added using `createNode` are provided a default set of node parameters. However, it is also possible to provide different parameters to each node using `MockNodeParameters`. Of particular interest are `configOverrides` which allow you to override some of the default node configuration options. Please refer to the `MockNodeConfigOverrides` class for details what can currently be overridden. Also, the `additionalCordapps` parameter allows you to add extra CorDapps to a specific node. This is useful when you wish for all nodes to load a common CorDapp but for a subset of nodes to load CorDapps specific to their role in the network.

## Running the network

When using a `MockNetwork`, you must be careful to ensure that all the nodes have processed all the relevant messages before making assertions about the result of performing some action. For example, if you start a flow to update the ledger but don't wait until all the nodes involved have processed all the resulting messages, your nodes' vaults may not be in the state you expect.

When `networkSendManuallyPumped` is set to `false`, you must manually initiate the processing of received messages. You manually process received messages as follows:

- `StartedMockNode.pumpReceive()` processes a single message from the node's queue
- `MockNetwork.runNetwork()` processes all the messages in every node's queue until there are no further messages to process

When `networkSendManuallyPumped` is set to `true`, nodes will automatically process the messages they receive. You can block until all messages have been processed using `MockNetwork.waitQuiescent()`.

**Warning:** If `threadPerNode` is set to `true`, `networkSendManuallyPumped` must also be set to `true`.

## Running flows

A `StartedMockNode` starts a flow using the `StartedNodeServices.startFlow` method. This method returns a future representing the output of running the flow.

```
val signedTransactionFuture = nodeA.services.startFlow(IOUFlow(iouValue = 99,  
    ↪otherParty = nodeBParty))
```

```
CordaFuture<SignedTransaction> future = startFlow(a.getServices(), new ExampleFlow.  
    ↪Initiator(1, nodeBParty));
```

The network must then be manually run before retrieving the future's value:

```
val signedTransactionFuture = nodeA.services.startFlow(IOUFlow(iouValue = 99,  
    ↪otherParty = nodeBParty))  
// Assuming network.networkSendManuallyPumped == false.  
network.runNetwork()  
val signedTransaction = future.get();
```

```
CordaFuture<SignedTransaction> future = startFlow(a.getServices(), new ExampleFlow.  
    ↪Initiator(1, nodeBParty));  
// Assuming network.networkSendManuallyPumped == false.  
network.runNetwork();  
SignedTransaction signedTransaction = future.get();
```

## Accessing `StartedMockNode` internals

### Querying a node's vault

Recorded states can be retrieved from the vault of a `StartedMockNode` using:

```
val myStates = nodeA.services.vaultService.queryBy<MyStateType>().states
```

```
List<MyStateType> myStates = node.getServices().getVaultService().queryBy(MyStateType.  
    ↪class).getStates();
```

This allows you to check whether a given state has (or has not) been stored, and whether it has the correct attributes.

### Examining a node's transaction storage

Recorded transactions can be retrieved from the transaction storage of a `StartedMockNode` using:

```
val transaction = nodeA.services.validatedTransactions.getTransaction(transaction.id)
```

```
SignedTransaction transaction = nodeA.getServices().getValidatedTransactions().  
    ↪getTransaction(transaction.getId())
```

This allows you to check whether a given transaction has (or has not) been stored, and whether it has the correct attributes.

This allows you to check whether a given state has (or has not) been stored, and whether it has the correct attributes.

## Further examples

- See the flow testing tutorial [here](#)
- See the oracle tutorial [here](#) for information on testing @CordaService classes
- Further examples are available in the Example CorDapp in [Java](#) and [Kotlin](#)

### 4.13.2 Contract testing

The Corda test framework includes the ability to create a test ledger by calling the `ledger` function on an implementation of the `ServiceHub` interface.

#### Test identities

You can create dummy identities to use in test transactions using the `TestIdentity` class:

```
val bigCorp = TestIdentity((CordaX500Name("BigCorp", "New York", "GB")))
```

```
private static final TestIdentity bigCorp = new TestIdentity(new CordaX500Name(
    "BigCorp", "New York", "GB"));
```

`TestIdentity` exposes the following fields and methods:

```
val identityParty: Party = bigCorp.party
val identityName: CordaX500Name = bigCorp.name
val identityPubKey: PublicKey = bigCorp.publicKey
val identityKeyPair: KeyPair = bigCorp.keyPair
val identityPartyAndCertificate: PartyAndCertificate = bigCorp.identity
```

```
Party identityParty = bigCorp.getParty();
CordaX500Name identityName = bigCorp.getName();
PublicKey identityPubKey = bigCorp.getPublicKey();
KeyPair identityKeyPair = bigCorp.getKeyPair();
PartyAndCertificate identityPartyAndCertificate = bigCorp.getIdentity();
```

You can also create a unique `TestIdentity` using the `fresh` method:

```
val uniqueTestIdentity: TestIdentity = TestIdentity.fresh("orgName")
```

```
TestIdentity uniqueTestIdentity = TestIdentity.Companion.fresh("orgName");
```

#### MockServices

A mock implementation of `ServiceHub` is provided in `MockServices`. This is a minimal `ServiceHub` that suffices to test contract logic. It has the ability to insert states into the vault, query the vault, and construct and check transactions.

```
private val ledgerServices = MockServices(
    // A list of packages to scan for cordapps
    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]
    megaCorp,
    mock<IdentityService>().also {
        doReturn(megaCorp.party).whenever(it).partyFromKey(megaCorp.publicKey)
        doReturn(null).whenever(it).partyFromKey(bigCorp.publicKey)
        doReturn(null).whenever(it).partyFromKey(alice.publicKey)
    })
}
```

```
ledgerServices = new MockServices(
    // A list of packages to scan for cordapps
    singletonList("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]
    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
    makeTestIdentityService(megaCorp.getIdentity())
);
```

Alternatively, there is a helper constructor which just accepts a list of `TestIdentity`. The first identity provided is the identity of the node whose `ServiceHub` is being mocked, and any subsequent identities are identities that the node knows about. Only the calling package is scanned for cordapps and a test `IdentityService` is created for you, using all the given identities.

```
@Suppress("unused")
private val simpleLedgerServices = MockServices(
    // This is the identity of the node
    megaCorp,
    // Other identities the test node knows about
    bigCorp,
    alice
)
```

```
private final MockServices simpleLedgerServices = new MockServices(
    // This is the identity of the node
    megaCorp,
    // Other identities the test node knows about
    bigCorp,
    alice
);
```

## Writing tests using a test ledger

The `ServiceHub.ledger` extension function allows you to create a test ledger. Within the ledger wrapper you can create transactions using the `transaction` function. Within a transaction you can define the `input` and `output` states for the transaction, alongside any commands that are being executed, the `timeWindow` in which the transaction has been executed, and any attachments, as shown in this example test:

```
@Test
fun simpleCPMoveSuccess() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            input(CP_PROGRAM_ID, inState)
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            attachments(CP_PROGRAM_ID)
            timeWindow(TEST_TX_TIME)
            output(CP_PROGRAM_ID, "alice's paper", inState.withOwner(alice.party))
            verifies()
        }
    }
}
```

```
@Test
public void simpleCPMoveSuccess() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.input(JCP_PROGRAM_ID, inState);
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Move());
            tx.attachments(JCP_PROGRAM_ID);
            tx.timeWindow(TEST_TX_TIME);
            tx.output(JCP_PROGRAM_ID, "alice's paper", inState.withOwner(alice.
        ↪getParty()));
            return tx.verifies();
        });
        return Unit.INSTANCE;
    });
}
```

Once all the transaction components have been specified, you can run `verifies()` to check that the given transaction is valid.

## Checking for failure states

In order to test for failures, you can use the `failsWith` method, or in Kotlin the `fails with` helper method, which assert that the transaction fails with a specific error. If you just want to assert that the transaction has failed without verifying the message, there is also a `fails` method.

```
@Test
fun simpleCPMoveFails() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            input(CP_PROGRAM_ID, inState)
```

(continues on next page)

(continued from previous page)

```
        command(megaCorp.publicKey, CommercialPaper.Commands.Move())
        attachments(CP_PROGRAM_ID)
        `fails with` ("the state is propagated")
    }
}
}
```

```
@Test
public void simpleCPMoveFails() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.input(JCP_PROGRAM_ID, inState);
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
→Move());
            tx.attachments(JCP_PROGRAM_ID);
            return txfailsWith("the state is propagated");
        });
        return Unit.INSTANCE;
    });
}
```

**Note:** The transaction DSL forces the last line of the test to be either a `verifies` or `fails` with statement.

## Testing multiple scenarios at once

Within a single transaction block, you can assert several times that the transaction constructed so far either passes or fails verification. For example, you could test that a contract fails to verify because it has no output states, and then add the relevant output state and check that the contract verifies successfully, as in the following example:

```
@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()
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        tx.attachments(JCP_PROGRAM_ID);
        txfailsWith("the state is propagated");
        tx.output(JCP_PROGRAM_ID, "alice's paper", inState.withOwner(alice.
        ↪getParty()));
        return tx.verifies();
    });
    return Unit.INSTANCE;
});
}

```

You can also use the `tweak` function to create a locally scoped transaction that you can make changes to and then return to the original, unmodified transaction. As in the following example:

```

@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()
    }
}

```

```

@Test
public void simpleIssuanceWithTweakTopLevelTx() {
    transaction(ledgerServices, tx -> {
        tx.output(JCP_PROGRAM_ID, "paper", getPaper()); // Some CP is issued onto the
        ↪ledger by MegaCorp.
        tx.attachments(JCP_PROGRAM_ID);
        tx.tweak(tw -> {
            tw.command(bigCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Issue());
            tw.timeWindow(TEST_TX_TIME);
            return tw.failsWith("output states are issued by a command signer");
        });
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.Issue());
        tx.timeWindow(TEST_TX_TIME);
        return tx.verifies();
    });
}

```

## Chaining transactions

The following example shows that within a ledger, you can create more than one transaction in order to test chains of transactions. In addition to `transaction`, `unverifiedTransaction` can be used, as in the example below, to create transactions on the ledger without verifying them, for pre-populating the ledger with existing data. When chaining transactions, it is important to note that even though a transaction verifies successfully, the

overall ledger may not be valid. This can be verified separately by placing a `verifies` or `fails` statement within the ledger block.

```
@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()
    }
}
```

```
@Test
public void chainCommercialPaperDoubleSpend() {
    PartyAndReference issuer = megaCorp.ref(defaultRef);
    ledger(ledgerServices, l -> {
        l.unverifiedTransaction(tx -> {
            tx.output(Cash.PROGRAM_ID, "alice's $900",
                new Cash.State(issuedBy(DOLLARS(900), issuer), alice.getParty()));
            tx.attachments(Cash.PROGRAM_ID);
            return Unit.INSTANCE;
        });
    });
}
```

(continues on next page)

(continued from previous page)

```
// Some CP is issued onto the ledger by MegaCorp.
1.transaction("Issuance", tx -> {
    tx.output(JCP_PROGRAM_ID, "paper", getPaper());
    tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
    ↪Issue());
    tx.attachments(JCP_PROGRAM_ID);
    tx.timeWindow(TEST_TX_TIME);
    return tx.verifies();
}) ;

1.transaction("Trade", tx -> {
    tx.input("paper");
    tx.input("alice's $900");
    tx.output(Cash.PROGRAM_ID, "borrowed $900", new Cash.
    ↪State(issuedBy(DOLLARS(900), issuer), megaCorp.getParty()));
    JavaCommercialPaper.State inputPaper = 1.
    ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
    tx.output(JCP_PROGRAM_ID, "alice's paper", inputPaper.withOwner(alice.
    ↪getParty()));
    tx.command(alice.getPublicKey(), new Cash.Commands.Move());
    tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
    ↪Move());
    return tx.verifies();
}) ;

1.transaction(tx -> {
    tx.input("paper");
    JavaCommercialPaper.State inputPaper = 1.
    ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
    // We moved a paper to other pubkey.
    tx.output(JCP_PROGRAM_ID, "bob's paper", inputPaper.withOwner(bob.
    ↪getParty()));
    tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
    ↪Move());
    return tx.verifies();
});
1fails();
return Unit.INSTANCE;
});
}
```

## Further examples

- See the flow testing tutorial [here](#)
- Further examples are available in the Example CorDapp in [Java](#) and [Kotlin](#)

Before reading this page, you should be familiar with the [key concepts of Corda](#).

## 4.14 API stability guarantees

Corda makes certain commitments about what parts of the API will preserve backwards compatibility as they change and which will not. Over time, more of the API will fall under the stability guarantees. Thus, APIs can be categorized in the following 2 broad categories:

- **public APIs**, for which API/ABI backwards compatibility guarantees are provided. See: [Public API](#)
- **non-public APIs**, for which no backwards compatibility guarantees are provided. See: [Non-public API \(experimental\)](#)

## 4.15 Public API

The following modules form part of Corda's public API and we commit to API/ABI backwards compatibility in following releases, unless an incompatible change is required for security reasons:

- **Core (net.corda.core)**: 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
- **Client RPC (net.corda.client.rpc)**: client RPC
- **Client Jackson (net.corda.client.jackson)**: JSON support for client applications
- **DSL Test Utils (net.corda.testing.dsl)**: a simple DSL for building pseudo-transactions (not the same as the wire protocol) for testing purposes.
- **Test Node Driver (net.corda.testing.node, net.corda.testing.driver)**: test utilities to run nodes programmatically
- **Test Utils (net.corda.testing.core)**: generic test utilities
- **Http Test Utils (net.corda.testing.http)**: a small set of utilities for making HttpCalls, aimed at demos and tests.
- **Dummy Contracts (net.corda.testing.contracts)**: dummy state and contracts for testing purposes
- **Mock Services (net.corda.testing.services)**: mock service implementations for testing purposes

## 4.16 Non-public API (experimental)

The following modules are not part of the Corda's public API and no backwards compatibility guarantees are provided. They are further categorized in 2 classes:

- the incubating modules, for which we will do our best to minimise disruption to developers using them until we are able to graduate them into the public API
- the internal modules, which are not to be used, and will change without notice

### 4.16.1 Corda incubating modules

- **net.corda.confidential**: experimental support for confidential identities on the ledger
- **net.corda.finance**: a range of elementary contracts (and associated schemas) and protocols, such as abstract fungible assets, cash, obligation and commercial paper
- **net.corda.client.jfx**: support for Java FX UI
- **net.corda.client.mock**: client mock utilities
- **Cordformation**: Gradle integration plugins

## 4.16.2 Corda internal modules

Everything else is internal and will change without notice, even deleted, and should not be used. This also includes any package that has `.internal` in it. So for example, `net.corda.core.internal` and sub-packages should not be used.

Some of the public modules may depend on internal modules, so be careful to not rely on these transitive dependencies. In particular, the testing modules depend on the node module and so you may end having the node in your test classpath.

**Warning:** The web server module will be removed in future. You should call Corda nodes through RPC from your web server of choice e.g., Spring Boot, Vertx, Undertow.

## 4.17 The `@DoNotImplement` annotation

Certain interfaces and abstract classes within the Corda API have been annotated as `@DoNotImplement`. While we undertake not to remove or modify any of these classes' existing functionality, the annotation is a warning that we may need to extend them in future versions of Corda. Cordapp developers should therefore just use these classes "as is", and *not* attempt to extend or implement any of them themselves.

This annotation is inherited by subclasses and sub-interfaces.

---

**CHAPTER  
FIVE**

---

**CHEAT SHEET**

A “cheat sheet” summarizing the key Corda types. A PDF version is downloadable [here](#).



## CORDA CHEAT SHEET

### Useful links:

Website:	<a href="http://corda.net">corda.net</a>
Github org.:	<a href="https://github.com/corda">github.com/corda</a>
Documentation:	<a href="http://docs.corda.net">docs.corda.net</a>
Slack:	<a href="https://slack.corda.net">slack.corda.net</a>

Stack Overflow: [stackoverflow.com/questions/tagged/corda](https://stackoverflow.com/questions/tagged/corda)

### RUNNING CORDA

#### a. Set up your dev environment

<https://docs.corda.net/getting-set-up.html>

#### b. Clone the template app in Kotlin or Java

git clone <https://github.com/corda/cordapp-template-kotlin>

#### c. Check out the latest version (e.g. V2)

cd cordapp-template-kotlin && git checkout release-V2

#### d. Deploy the nodes

./gradlew clean deployNodes

#### e. Run the nodes

Unix: sh kotlin-source/build/nodes/runnodes

Windows: call kotlin-source/build/nodes/runnodes.bat

### STATES

#### ContractState

The base class for on-ledger states

#### .participants

The parties for which this state is relevant

#### LinearState (extends ContractState)

State representing a 'shared fact' evolving over time

#### .linearId

An ID shared by all evolutions of the 'shared fact'

#### OwnableState (extends ContractState)

State representing fungible assets (cash, oil...)

#### .owner

The state's current owner

#### .withNewOwner(AbstractParty)

Creates a copy of the state with a new owner

### CONTRACTS

#### Contract

Establishes which transactions are valid for a given state

#### .verify(LedgerTransaction)

Throws an exception if the transaction is invalid

### TRANSACTIONS

#### TransactionBuilder

A mutable container for building a general transaction

#### .withItems(vararg Any)

Adds items (states, commands...) to the builder

#### ServiceHub.signInitialTransaction(TransactionBuilder)

Converts the builder to a signed transaction

### TRANSACTIONS (CONT.)

#### SignedTransaction

An immutable transaction plus its associated digital signatures

#### .verifyRequiredSignatures()

Verify all the transaction's required signatures

#### .verifySignaturesExcept(vararg List<PublicKey>)

Verify all the transaction's required signatures except those listed

#### .verify(ServiceHub, boolean)

Verify the transaction

#### .toLedgerTransaction(ServiceHub, boolean)

Resolve transaction into a LedgerTransaction for extra verification

#### ServiceHub.addSignature(SignedTransaction)

Add a digital signature to the transaction

### FLOWS

#### FlowLogic

The actions executed by one side of a flow

#### .initiateFlow(Party)

Initiates communication between two flows

#### FlowSession.send(Party, Any)/FlowSession.receive(Party)

Sends data to/receives data from the specified counterparty

#### .subFlow(FlowLogic-R, Boolean)

Invokes a sub-flow that may return a result

#### .serviceHub

Provides access to the node's services

### FLOW ANNOTATIONS

#### @InitiatingFlow

A flow that is started directly

#### @InitiatedBy(KClass)

A flow that is only started by a message from an InitiatingFlow

#### @StartableByRPC

Allows the flow to be started via RPC by the node's owner

### SERVICE HUB

#### .networkMapCache

Provides info on other nodes on the network (e.g. notaries...)

#### .vaultService

Stores the node's current and historic states

#### .validatedTransactions

Stores all the transactions seen by the node

#### .keyManagementService

Manages the node's digital signing keys

#### .myInfo

Other information about the node

#### .clock

Provides access to the node's internal time and date

### PROVIDING AN API

#### a. Subclass WebServerPluginRegistry

```
class MyWebPlugin : WebServerPluginRegistry() { ... }
```

129

#### b. Override webApis

```
override val webApis = listOf(Function(::MyApi))
```

#### c. Register the fully qualified class name of the plugin

```
...under src/main/resources/META-INF/services/WebPluginRegistry
```

## GETTING STARTED DEVELOPING CORDAPPS

### 6.1 Running the example CorDapp

At this point we've set up the development environment, and have an example CorDapp in an IntelliJ project. In this section, the CorDapp will be deployed to locally running Corda nodes.

The local Corda network includes one notary, and three nodes, each representing parties in the network. A Corda node is an individual instance of Corda representing one party in a network. For more information on nodes, see the [node documentation](#).

Before continuing, ensure that you've [set up your development environment](#).

#### 6.1.1 Step One: Deploy the CorDapp locally

The first step is to deploy the CorDapp to nodes running locally.

1. Navigate to the root directory of the example CorDapp.
2. To deploy the nodes on Windows run the following command: `gradlew clean deployNodes`  
To deploy the nodes on Mac or Linux run the following command: `./gradlew clean deployNodes`
3. To best understand the deployment process, there are several perspectives it is helpful to see. On Windows run the following command: `workflows-kotlin\build\nodes\runnodes`  
On Mac/Linux run the following command: `workflows-kotlin/build/nodes/runnodes`

This command opens four terminal windows: the notary, and a node each for PartyA, PartyB, and PartyC. A notary is a validation service that prevents double-spending, enforces timestamping, and may also validate transactions. For more information on notaries, see the [notary documentation](#).

---

**Note:** Maintain window focus on the node windows, if the nodes fail to load, close them using `ctrl + d`. The `runnodes` script opens each node directory and runs `java -jar corda.jar`.

---

4. Go through the tabs to see the perspectives of other network members.

#### 6.1.2 Step Two: Run a CorDapp transaction

1. Open the terminal window for PartyA. From this window, any flows executed will be from the perspective of PartyA.

2. To execute the `ExampleFlow.kt` flow, run the following command: `flow start ExampleFlow iouValue: 1, otherParty: PartyB`

A flow is the mechanism by which a transaction takes place using Corda. This flow creates an instance of the IOU state, which requires an `iouValue` property. Flows are contained in CorDapps, and define the mechanisms by which parties transact. For more information on flows, see the [flow documentation](#).

3. To check whether PartyB has received the transaction, open the terminal window showing PartyB's perspective, and run the following command: `run vaultQuery contractStateType: com.example.state.IOUState`

This command displays all of the IOU states in the node's vault. States are immutable objects that represent shared facts between the parties. States serve as the inputs and outputs of transactions.

### 6.1.3 Next steps

After deploying the example CorDapp, the next step is to start [writing a CorDapp](#) containing your own contract, states, and flows.

## 6.2 Building your own CorDapp

After examining a functioning CorDapp, the next challenge is to create one of your own. We're going to build a simple supply chain CorDapp representing a network between a car dealership, a car manufacturer, and a bank.

To model this network, you need to create one state (representing cars), one contract (to control the rules governing cars), and one flow (to create cars). This CorDapp will be very basic, but entirely functional and deployable.

### 6.2.1 Step One: Download a template CorDapp

The first thing you need to do is clone a CorDapp template to modify.

1. Open a terminal and navigate to a directory to store the new project.
2. Run the following command to clone the template CorDapp: `git clone https://github.com/corda/cordapp-template-kotlin.git`
3. Open IntelliJ and open the CorDapp template project.
4. Click **File > Project Structure**. To set the project SDK click **New... > JDK**, and navigating to the installation directory of your JDK. Click **Apply**.
5. Select **Modules > + > Import Module**. Select the `cordapp-template-kotlin` folder and click **Open**. Select **Import module from external model > Gradle > Next** > tick the **Use auto-import** checkbox > **Finish > Ok**. Gradle will now download all the project dependencies and perform some indexing.

### 6.2.2 Step Two: Creating states

Since the CorDapp models a car dealership network, a state must be created to represent cars. States are immutable objects representing on-ledger facts. A state might represent a physical asset like a car, or an intangible asset or agreement like an IOU. For more information on states, see the [state documentation](#).

1. From IntelliJ expand the source files and navigate to the following state template file: `contracts > src > main > kotlin > com.template > states > TemplateState.kt`.
2. Right-click on `TemplateState.kt` in the project navigation on the left. Select **Refactor > Copy**.

3. Rename the file to CarState and click **OK**.
4. Double-click the new state file to open it. Add the following imports to the top of the state file:

```
package com.template.states

import com.template.contracts.CarContract
import com.template.contracts.TemplateContract
import net.corda.core.contracts.BelongsToContract
import net.corda.core.contracts.ContractState
import net.corda.core.contracts.UniqueIdentifier
import net.corda.core.identity.AbstractParty
import net.corda.core.identity.Party
```

It's important to specify what classes are required in each state, contract, and flow. This process must be repeated with each file as it is created.

5. Update `@BelongsToContract (TemplateContract::class)` to specify `CarContract::class`.
6. Add the following fields to the state:

- `owningBank` of type `Party`
- `holdingDealer` of type `Party`
- `manufacturer` of type `Party`
- `vin` of type `String`
- `licensePlateNumber` of type `String`
- `make` of type `String`
- `model` of type `String`
- `dealershipLocation` of type `String`
- `linearId` of type `UniqueIdentifier`

Don't worry if you're not sure exactly how these should appear, you can check your code shortly.

7. Remove the `data` and `participants` parameters.
8. Add a body to the `CarState` class that overrides `participants` to contain a list of `owningBank`, `holdingDealer`, and `manufacturer`.
9. The `CarState` file should now appear as follows:

```
package com.template.states

import com.template.contracts.CarContract
import com.template.contracts.TemplateContract
import net.corda.core.contracts.BelongsToContract
import net.corda.core.contracts.ContractState
import net.corda.core.contracts.UniqueIdentifier
import net.corda.core.identity.AbstractParty
import net.corda.core.identity.Party

// *****
// * State *
// *****

@BelongsToContract(CarContract::class)
data class CarState(
```

(continues on next page)

(continued from previous page)

```

    val owningBank: Party,
    val holdingDealer: Party,
    val manufacturer: Party,
    val vin: String,
    val licensePlateNumber: String,
    val make: String,
    val model: String,
    val dealershipLocation: String,
    val linearId: UniqueIdentifier
) : ContractState {
    override val participants: List<AbstractParty> = listOf(owningBank, ^
    holdingDealer, manufacturer)
}

```

10. Save the `CarState.kt` file.

The `CarState` definition has now been created. It lists the properties and associated types required of all instances of this state.

### 6.2.3 Step Three: Creating contracts

After creating a state, you must create a contract. Contracts define the rules that govern how states can be created and evolved. For example, a contract for a Cash state should check that any transaction that changes the ownership of the cash is signed by the current owner and does not create cash from thin air. To learn more about contracts, see the [contracts documentation](#).

1. From IntelliJ, expand the project source and navigate to: `contracts > src > main > kotlin > com > template > contracts > TemplateContract.kt`
2. Right-click on `TemplateContract.kt` in the project navigation on the left. Select **Refactor > Copy**.
3. Rename the file to `CarContract` and click **OK**.
4. Double-click the new contract file to open it.
5. Add the following imports to the top of the file:

```

package com.template.contracts

import com.template.states.CarState
import net.corda.core.contracts.CommandData
import net.corda.core.contracts.Contract
import net.corda.core.contracts.requireSingleCommand
import net.corda.core.contracts.requireThat
import net.corda.core.transactions.LedgerTransaction

```

6. Update the class name to: `CarContract`
7. Replace `const val ID = "com.template.contracts.TemplateContract"` with `val ID = CarContract::class.qualifiedName!!`. This ID field is used to identify contracts when building a transaction. This ID declaration ensures that the contract name is created dynamically and can simplify code refactoring.
8. Update the `Action` command to an `Issue` command. This represents an issuance of an instance of the `CarState` state.

Commands are the operations that can be performed on a state. A contract will often define command logic for several operations that can be performed on the state in question, for example, issuing a state, changing ownership, and marking the state retired.

9. Add `val command = tx.commands.requireSingleCommand<Commands>().value` at the beginning of the `verify()` method. The `verify()` method defines the verification rules that commands must satisfy to be valid.
10. The final function of the contract is to prevent unwanted behaviour during the flow. After the `val command = tx.commands...` line, add the following requirement code:

```
when(command) {
    is Commands.Issue -> requireThat {
        "There should be no input state" using (tx.inputs.isEmpty())
    }
}
```

11. Inside the `requireThat` block add additional lines defining the following requirements:

- There should be one output state.
- The output state must be of the type `CarState`.
- The `licensePlateNumber` must be seven characters long.

12. The `CarContract.kt` file should look as follows:

```
package com.template.contracts

import com.template.states.CarState
import net.corda.core.contracts.CommandData
import net.corda.core.contracts.Contract
import net.corda.core.contracts.requireSingleCommand
import net.corda.core.contracts.requireThat
import net.corda.core.transactions.LedgerTransaction

class CarContract : Contract {
    companion object {
        const val ID = "com.template.contracts.CarContract"
    }

    override fun verify(tx: LedgerTransaction) {

        val command = tx.commands.requireSingleCommand<Commands>().value

        when(command) {
            is Commands.Issue -> requireThat {
                "There should be no input state" using (tx.inputs.isEmpty())
                "There should be one input state" using (tx.outputs.size == 1)
                "The output state must be of type CarState" using (tx.outputs.get(0).`data` is CarState)
                val outputState = tx.outputs.get(0).`data` as CarState
                "The licensePlateNumber must be seven characters long" using (outputState.licensePlateNumber.length == 7)
            }
        }

        interface Commands : CommandData {
            class Issue : Commands
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
    }
}
```

13. Save the `CarContract.kt` file. The contract file now defines rules that all transactions creating car states must follow.

#### 6.2.4 Step Four: Creating a flow

1. From IntelliJ, expand the project source and navigate to: `workflows > src > main > kotlin > com.template.flows > Flows.kt`
2. Right-click on **Flows.kt** in the project navigation on the left. Select **Refactor > Copy**.
3. Rename the file to `CarFlow` and click **OK**.
4. Add the following imports to the top of the file:

```
package com.template.flows

import co.paralleluniverse.fibers.Suspendable
import com.template.contracts.CarContract
import com.template.states.CarState
import net.corda.core.contracts.Command
import net.corda.core.contracts.UniqueIdentifier
import net.corda.core.contracts.requireThat
import net.corda.core.flows.*
import net.corda.core.identity.Party
import net.corda.core.node.ServiceHub
import net.corda.core.transactions.SignedTransaction
import net.corda.core.transactions.TransactionBuilder
```

5. Double-click the new contract file to open it.
6. Update the name of the `Initiator` class to `CarIssueInitiator`.
7. Update the name of the `Responder` class to `CarIssueResponder`.
8. Update the `@InitiatedBy` property of `CarIssueResponder` to `CarIssueInitiator::class`.
9. Now that the flow structure is in place, we can begin writing the code to create a transaction to issue a car state. Add parameters to the `CarIssueInitiator` class for all the fields of the `CarState` definition, except for `linearId`.
10. Inside the `call()` function of the initiator, create a variable for the notary node: `val notary = serviceHub.networkMapCache.notaryIdentities.first()`

**Note:** The `networkMapCache` contains information about the nodes and notaries inside the network.

11. Create a variable for an `Issue` command.

The first parameter of the command must be the command type, in this case `Issue`. As discussed above, the command tells other nodes what the purpose of the transaction is.

The second parameter of the command must be a list of keys from the relevant parties, in this case `owningBank`, `holdingDealer`, and `manufacturer`. As well as informing parties what the purpose of the transaction is, the command also specifies which signatures must be present on the associated transaction in order for it to be valid.

12. Create a `CarState` object using the parameters of `CarIssueInitiator`.

The last parameter for `CarState` must be a new `UniqueIdentifier()` object.

13. The `CarFlow.kt` file should look like this:

```
@InitiatingFlow
@StartableByRPC
class CarIssueInitiator(
    val owningBank: Party,
    val holdingDealer: Party,
    val manufacturer: Party,
    val vin: String,
    val licensePlateNumber: String,
    val make: String,
    val model: String,
    val dealershipLocation: String
) : FlowLogic<Unit>() {

    @Suspendable
    override fun call() {
        val notary = serviceHub.networkMapCache.notaryIdentities.first()
        val command = Command(CarContract.Commands.Issue(), listOf(owningBank, holdingDealer, manufacturer).map { it.owningKey })
        val carState = CarState(owningBank, holdingDealer, manufacturer, vin, licensePlateNumber, make, model, dealershipLocation, UniqueIdentifier())
    }
}

@InitiatedBy(CarIssueInitiator::class)
class CarIssueResponder(val counterpartySession: FlowSession) : FlowLogic<Unit>() {
    ...
    @Suspendable
    override fun call() {
        ...
    }
}
```

14. Update the `FlowLogic<Unit>` to `FlowLogic<SignedTransaction>` in both the initiator and responder class. This indicates that the `SignedTransaction` produced by this flow is returned from `call` and sent to the caller of the flow. If left unchanged, `FlowLogic<Unit>` will return nothing.

15. Update the return type of both `call()` transactions to be of type `SignedTransaction`.

16. In the `call()` function, create a `TransactionBuilder` object similarly. The `TransactionBuilder` class should take in the notary node. The output state and command must be added to the `TransactionBuilder`.

17. Verify the transaction by calling `verify(serviceHub)` on the `TransactionBuilder`.

18. Sign the transaction and store the result in a variable, using the following `serviceHub` method:

```
val notary = serviceHub.networkMapCache.notaryIdentities.first()
```

19. Delete the `progressTracker` as it won't be used in this tutorial.

20. The `CarFlow.kt` file should now look like this:

```

@InitiatingFlow
@StartableByRPC
class CarIssueInitiator(
    val owningBank: Party,
    val holdingDealer: Party,
    val manufacturer: Party,
    val vin: String,
    val licensePlateNumber: String,
    val make: String,
    val model: String,
    val dealershipLocation: String
) : FlowLogic<SignedTransaction>() {

    @Suspendable
    override fun call(): SignedTransaction {

        val notary = serviceHub.networkMapCache.notaryIdentities.first()
        val command = Command(CarContract.Commands.Issue(), listOf(owningBank,_
        ↪holdingDealer, manufacturer).map { it.owningKey })
        val carState = CarState(
            owningBank,
            holdingDealer,
            manufacturer,
            vin,
            licensePlateNumber,
            make,
            model,
            dealershipLocation,
            UniqueIdentifier()
        )

        val txBuilder = TransactionBuilder(notary)
            .addOutputState(carState, CarContract.ID)
            .addCommand(command)

        txBuilder.verify(serviceHub)
        val tx = serviceHub.signInitialTransaction(txBuilder)
    }
}

@InitiatedBy(CarIssueInitiator::class)
class CarIssueResponder(val counterpartySession: FlowSession) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {

    }
}
}

```

21. To finish the initiator's `call()` function, other parties must sign the transaction. Add the following code to send the transaction to the other relevant parties:

```

val sessions = (carState.participants - ourIdentity).map { initiateFlow(it as_
↪Party) }
val stx = subFlow(CollectorsFlow(tx, sessions))
return subFlow(FinalityFlow(stx, sessions))

```

The first line creates a `List<FlowSession>` object by calling `initiateFlow()` for each party other than the initiating party. The second line collects signatures from the relevant parties and returns a signed transaction. The third line calls `FinalityFlow()`, finalizes the transaction using the notary or notary pool.

---

**Note:** Sessions are used for sending and receiving objects between nodes. `ourIdentity` is removed from the list of participants to open sessions to because a session does not need to be opened to the initiating party.

---

22. Lastly, the body of the responder flow must be completed. The following code checks the transaction contents, signs it, and sends it back to the initiator:

```
@Suspendable
override fun call(): SignedTransaction {
    val signedTransactionFlow = object : SignTransactionFlow(counterpartySession) {
        override fun checkTransaction(stx: SignedTransaction) = requireThat {
            val output = stx.tx.outputs.single().data
            "The output must be a CarState" using (output is CarState)
        }
        val txWeJustSignedId = subFlow(signedTransactionFlow)
        return subFlow(ReceiveFinalityFlow(counterpartySession, txWeJustSignedId.id))
    }
}
```

---

**Note:** The `checkTransaction` function should be used *only* to model business logic. A contract's `verify` function should be used to define what is and is not possible within a transaction.

---

23. The completed `CarFlow.kt` should look like this:

```
package com.template.flows

import co.paralleluniverse.fibers.Suspendable
import com.template.contracts.CarContract
import com.template.states.CarState
import net.corda.core.contracts.Command
import net.corda.core.contracts.UniqueIdentifier
import net.corda.core.contracts.requireThat
import net.corda.core.flows.*
import net.corda.core.identity.Party
import net.corda.core.node.ServiceHub
import net.corda.core.transactions.SignedTransaction
import net.corda.core.transactions.TransactionBuilder

@InitiatingFlow
@StartableByRPC
class CarIssueInitiator(
    val owningBank: Party,
    val holdingDealer: Party,
    val manufacturer: Party,
    val vin: String,
    val licensePlateNumber: String,
    val make: String,
    val model: String,
```

(continues on next page)

(continued from previous page)

```

    val dealershipLocation: String
) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {

        val notary = serviceHub.networkMapCache.notaryIdentities.first()
        val command = Command(CarContract.Commands.Issue(), listOf(owningBank,
            holdingDealer, manufacturer).map { it.owningKey })
        val carState = CarState(
            owningBank,
            holdingDealer,
            manufacturer,
            vin,
            licensePlateNumber,
            make,
            model,
            dealershipLocation,
            UniqueIdentifier()
        )

        val txBuilder = TransactionBuilder(notary)
            .addOutputState(carState, CarContract.ID)
            .addCommand(command)

        txBuilder.verify(serviceHub)
        val tx = serviceHub.signInitialTransaction(txBuilder)

        val sessions = (carState.participants - ourIdentity).map {_
            initiateFlow(it as Party) }
        val stx = subFlow(CollectorsFlow(tx, sessions))
        return subFlow(FinalityFlow(stx, sessions))
    }
}

@InitiatedBy(CarIssueInitiator::class)
class CarIssueResponder(val counterpartySession: FlowSession) : FlowLogic
<SignedTransaction>() {

    @Suspendable
    override fun call(): SignedTransaction {
        val signedTransactionFlow = object :_
        SignTransactionFlow(counterpartySession) {
            override fun checkTransaction(stx: SignedTransaction) = requireThat {
                val output = stx.tx.outputs.single().data
                "The output must be a CarState" using (output is CarState)
            }
        }
        val txWeJustSignedId = subFlow(signedTransactionFlow)
        return subFlow(ReceiveFinalityFlow(counterpartySession, txWeJustSignedId.
            id))
    }
}

```

## 6.2.5 Step Five: Update the Gradle build

The Gradle build files must be updated to change the node configuration.

1. Navigate to the `build.gradle` file in the root `cordapp-template-kotlin` directory.
2. In the `deployNodes` task, update the nodes to read as follows:

```
node {
    name "O=Notary,L=London,C=GB"
    notary = [validating : false]
    p2pPort 10002
    rpcSettings {
        address("localhost:10003")
        adminAddress("localhost:10043")
    }
}
node {
    name "O=Dealership,L=London,C=GB"
    p2pPort 10005
    rpcSettings {
        address("localhost:10006")
        adminAddress("localhost:10046")
    }
    rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL
→"] ]]
}
node {
    name "O=Manufacturer,L=New York,C=US"
    p2pPort 10008
    rpcSettings {
        address("localhost:10009")
        adminAddress("localhost:10049")
    }
    rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL
→"] ]]
}
node {
    name "O=BankofAmerica,L=New York,C=US"
    p2pPort 10010
    rpcSettings {
        address("localhost:10007")
        adminAddress("localhost:10047")
    }
    rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL
→"] ]]
}
```

The `nodeDefaults` defines what CorDApps are installed on the nodes by default. To install additional CorDApps on the nodes, update the `nodeDefaults` definition, or add the CorDApps to each node definition individually.

3. Save the updated `build.gradle` file.

## 6.2.6 Step Six: Deploying your CorDapp locally

Now that the CorDapp code has been completed and the build file updated, the CorDapp can be deployed.

1. Open a terminal and navigate to the root directory of the project.
2. To deploy the nodes on Windows run the following command: `gradlew clean deployNodes`

To deploy the nodes on Mac or Linux run the following command: `./gradlew clean deployNodes`

3. To start the nodes on Windows run the following command: `build\nodes\runnodes`

To start the nodes on Mac/Linux run the following command: `build/nodes/runnodes`

---

**Note:** Maintain window focus on the node windows, if the nodes fail to load, close them using `ctrl + d`. The `runnodes` script opens each node directory and runs `java -jar corda.jar`.

---

4. To run flows in your CorDapp, enter the following flow command from any non-notary terminal window:

```
``flow start CarIssueInitiator owningBank: BankofAmerica, holdingDealer:_
  ↪Dealership, manufacturer: Manufacturer, vin: "abc", licensePlateNumber:_
  ↪"abc1234", make: "Honda", model: "Civic", dealershipLocation: "NYC"``
```

5. To check that the state was correctly issued, query the node using the following command:

```
run vaultQuery contractStateType: com.template.states.CarState
```

The vault is the node's repository of all information from the ledger that involves that node, stored in a relational model. After running the query, the terminal should display the state created by the flow command. This command can be run from the terminal window of any non-notary node, as all parties are participants in this transaction.

## 6.2.7 Next steps

The getting started experience is designed to be lightweight and get to code as quickly as possible, for more detail, see the following documentation:

- [CorDapp design best practice](#)
- [Testing CorDapp contracts](#)

For operational users, see the following documentation:

- [Node structure and configuration](#)
- [Deploying a node to a server](#)
- [Notary documentation](#)

Getting started with Corda will walk you through the process of setting up a development environment, deploying an example CorDapp, and building your own CorDapp based on the example.

1. [Setting up a development environment](#)
2. [Deploying an example CorDapp](#)
3. [Building your own CorDapp](#)

## 6.3 Setting up a development environment

### 6.3.1 Prerequisites

- **Java 8 JVK** - We require at least version 8u171, but do not currently support Java 9 or higher.

- **IntelliJ IDEA** - IntelliJ is an IDE that offers strong support for Kotlin and Java development. We support versions **2017.x**, **2018.x** and **2019.x** (with Kotlin plugin version 1.2.71).
- **Git** - We use Git to host our sample CorDapp and provide version control.

### 6.3.2 Step One: Downloading a sample project

1. Open a command prompt or terminal.
2. Clone the CorDapp example repo by running: `git clone https://github.com/corda/samples`
3. Move into the `cordapp-example` folder by running: `cd samples/cordapp-example`
4. Checkout the corresponding branch by running: `git checkout release-V4` in the current directory.

### 6.3.3 Step Two: Creating an IntelliJ project

1. Open IntelliJ. From the splash screen, click **Open**, navigate to and select the `cordapp-example` folder, and click **Ok**. This creates an IntelliJ project to work from.
2. Click **File > Project Structure**. To set the project SDK click **New... > JDK**, and navigating to the installation directory of your JDK. Click **Apply**.
3. Select **Modules > + > Import Module**. Select the `cordapp-example` folder and click **Open**. Select **Import module from external model > Gradle > Next** > tick the **Use auto-import** checkbox > **Finish > Ok**. Gradle will now download all the project dependencies and perform some indexing.

Your CorDapp development environment is now complete.

## 6.4 Next steps

Now that you've successfully set up your CorDapp development environment, we'll cover deploying an example CorDapp locally, before writing a CorDapp from scratch.

## 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:

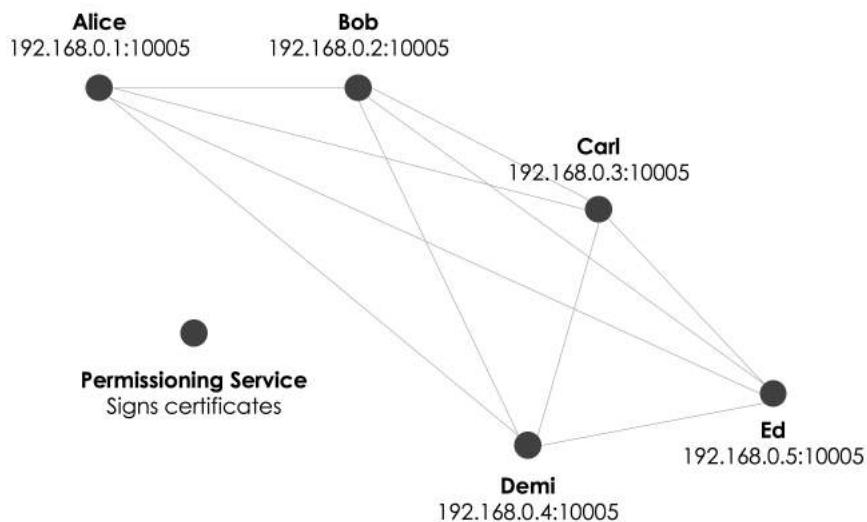
### 7.1 The network

#### Summary

- A Corda network is made up of nodes running Corda and CorDapps
- Communication between nodes is point-to-point, instead of relying on global broadcasts
- Each node has a certificate mapping their network identity to a real-world legal identity
- The network is permissioned, with access requiring a certificate from the network operator

#### 7.1.1 Network structure

A Corda network is a peer-to-peer network of **nodes**. Each node runs the Corda software as well as Corda applications known as **CorDapps**.



All communication between nodes is point-to-point and encrypted using transport-layer security. This means that data is shared only on a need-to-know basis. There are **no global broadcasts**.

### 7.1.2 Identity

Each node has a single well-known identity. The node's identity is used to represent the node in transactions, such as when purchasing an asset.

---

**Note:** These identities are distinct from the RPC user logins that are able to connect to the node via RPC.

---

Each network has a **network map service** that maps each well-known node identity to an IP address. These IP addresses are used for messaging between nodes.

Nodes can also generate confidential identities for individual transactions. The certificate chain linking a confidential identity to a well-known node identity or real-world legal identity is only distributed on a need-to-know basis. This ensures that even if an attacker gets access to an unencrypted transaction, they cannot identify the transaction's participants without additional information if confidential identities are being used.

### 7.1.3 Admission to the network

Corda networks are semi-private. To join a network, a node must obtain a certificate from the network operator. This certificate maps a well-known node identity to:

- A real-world legal identity
- A public key

The network operator enforces rules regarding the information that nodes must provide and the know-your-customer processes they must undergo before being granted this certificate.

## 7.2 The ledger

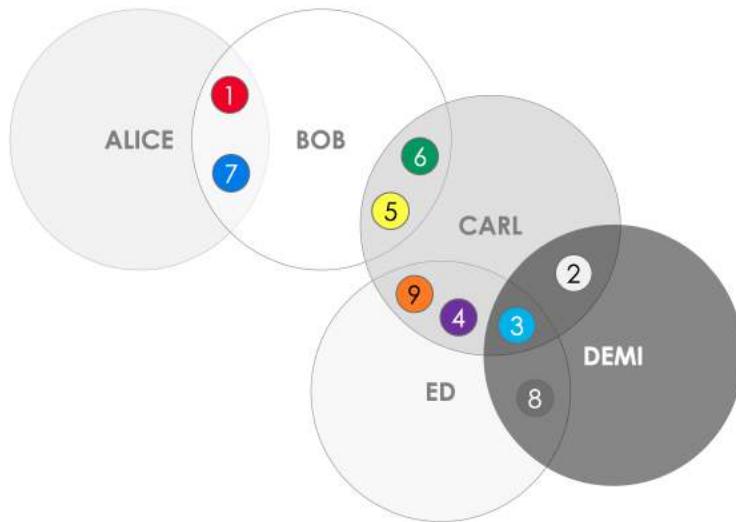
### Summary

- *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*

### 7.2.1 Overview

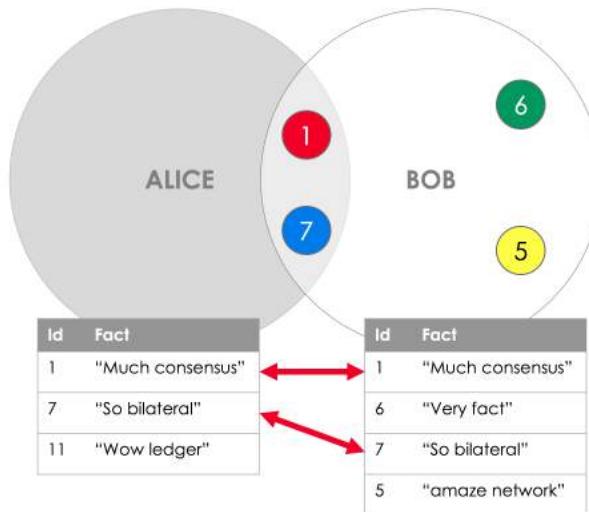
In Corda, there is **no single central store of data**. Instead, each node maintains a separate database of known facts. As a result, each peer only sees a subset of facts on the ledger, and no peer is aware of the ledger in its entirety.

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.

## 7.3 States

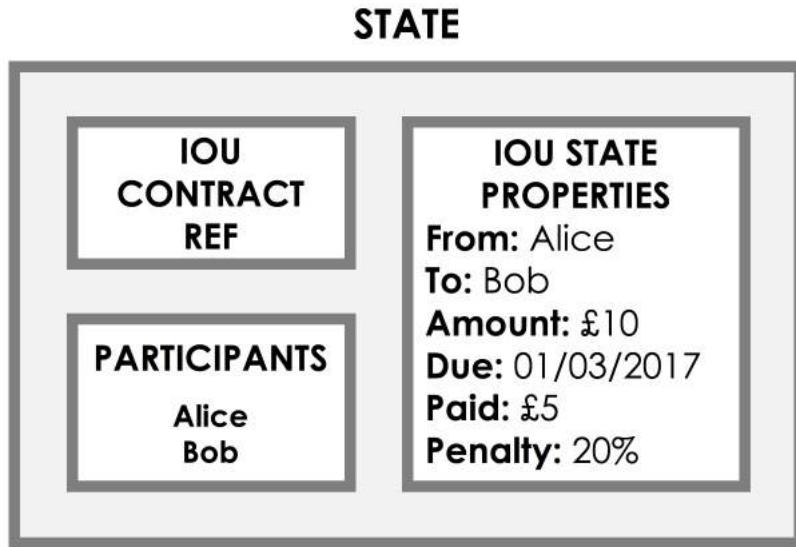
### Summary

- 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

### 7.3.1 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.

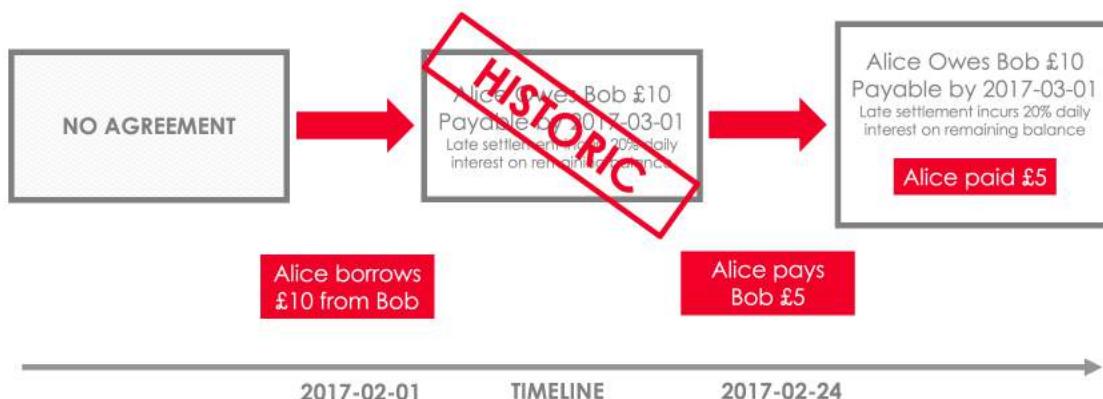
As well as any information about the fact itself, the state also contains a reference to the *contract* that governs the evolution of the state over time. We discuss contracts in [Contracts](#).

### 7.3.2 State sequences

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

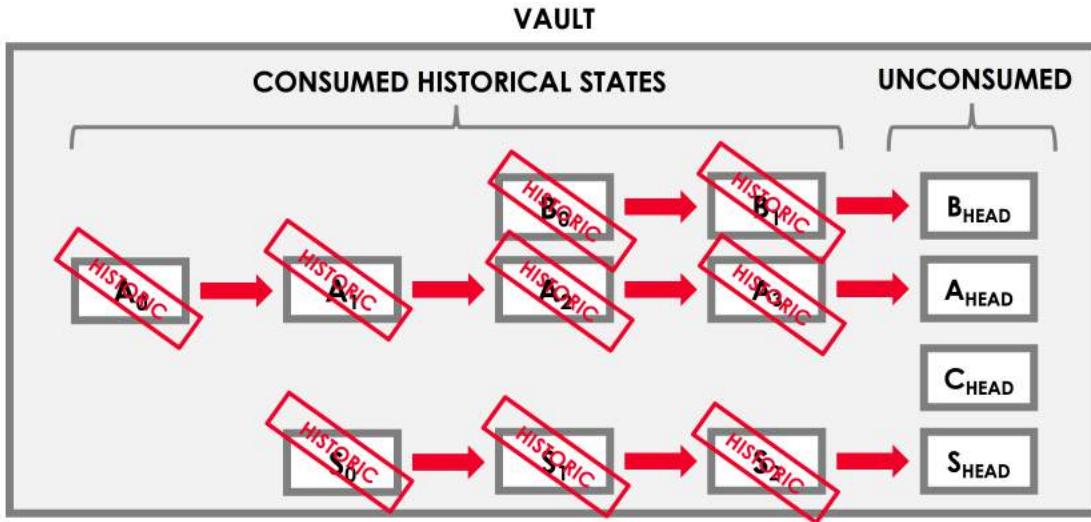
Instead, the lifecycle of a shared fact over time is represented by a **state sequence**. When a state needs to be updated, we create a new version of the state representing the new state of the world, and mark the existing state as historic.

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:



### 7.3.3 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.

### 7.3.4 Reference states

Not all states need to be updated by the parties which use them. In the case of reference data, there is a common pattern where one party creates reference data, which is then used (but not updated) by other parties. For this use-case, the states containing reference data are referred to as “reference states”. Syntactically, reference states are no different to regular states. However, they are treated different by Corda transactions. See [Transactions](#) for more details.

## 7.4 Transactions

### Summary

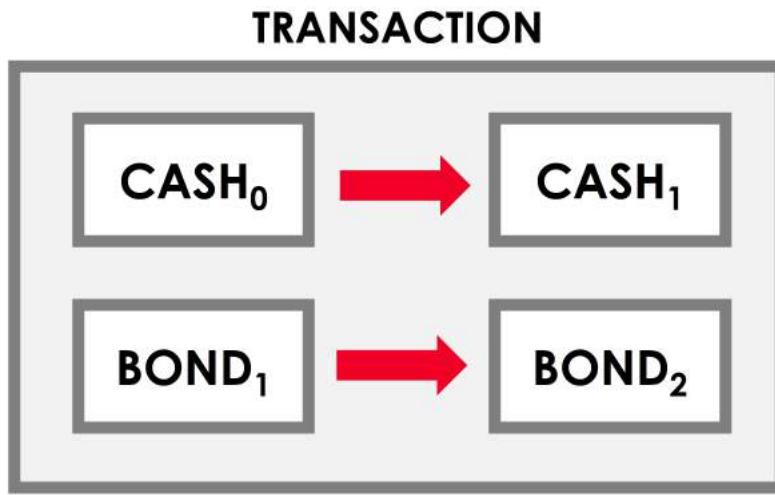
- *Transactions are proposals to update the ledger*
- *A transaction proposal will only be committed if:*
  - *It doesn't contain double-spends*
  - *It is contractually valid*
  - *It is signed by the required parties*

### 7.4.1 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, outputs and references 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)

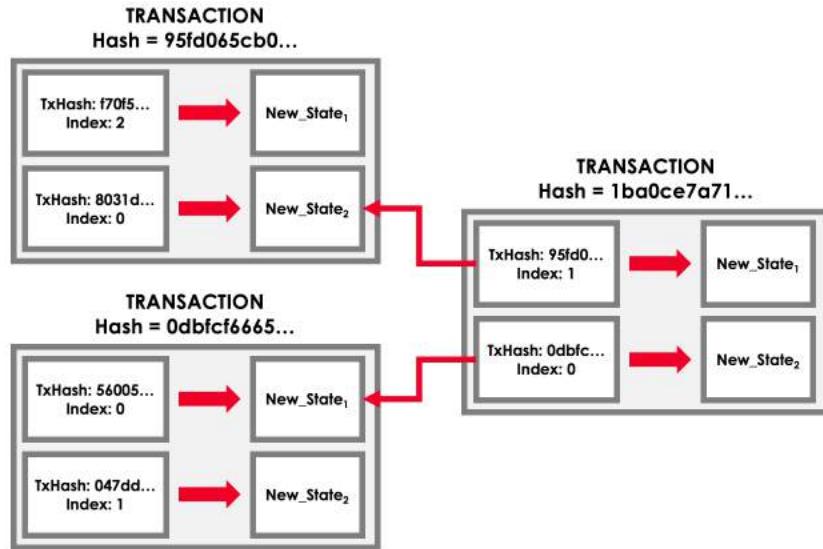
## 7.4.2 Transaction chains

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

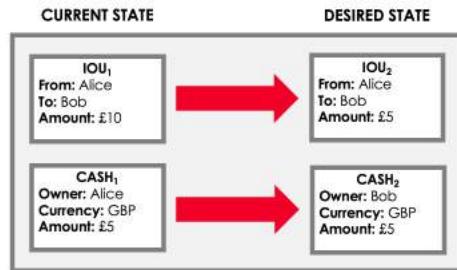
This situation can be illustrated as follows:



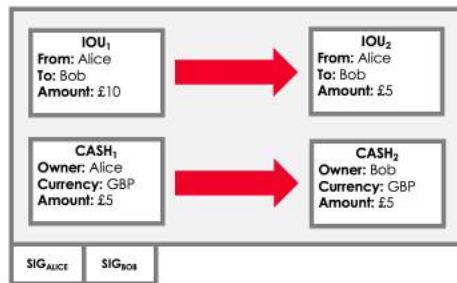
These input state references link together transactions over time, forming what is known as a *transaction chain*.

### 7.4.3 Committing transactions

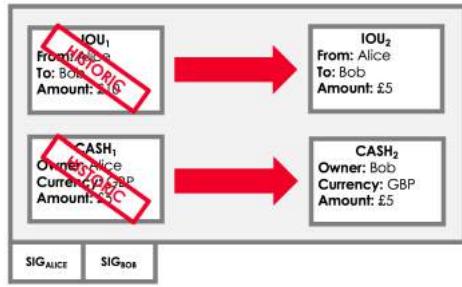
Initially, a transaction is just a **proposal** to update the ledger. It represents the future state of the ledger that is desired by the transaction builder(s):



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

#### 7.4.4 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.

#### 7.4.5 Reference states

As mentioned in [States](#), some states need to be referred to by the contracts of other input or output states but not updated/consumed. This is where reference states come in. When a state is added to the references list of a transaction, instead of the inputs or outputs list, then it is treated as a *reference state*. There are two important differences between regular states and reference states:

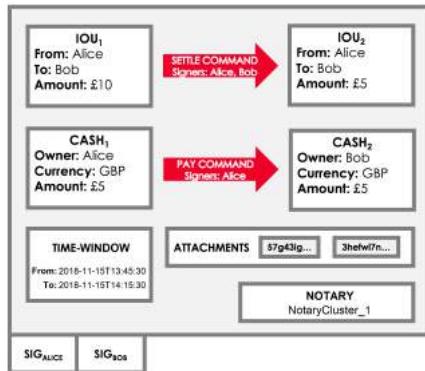
- The specified notary for the transaction **does** check whether the reference states are current. However, reference states are not consumed when the transaction containing them is committed to the ledger.
- The contracts for reference states are not executed for the transaction containing them.

#### 7.4.6 Other transaction components

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

- Commands
- Attachments
- Time-Window
- Notary

For example, suppose we have a transaction where Alice uses a £5 cash payment to pay off £5 of an IOU with Bob. This transaction has two supporting attachments and will only be notarised by NotaryClusterA if the notary pool receives it within the specified time-window. This transaction would look as follows:



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

## Commands

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

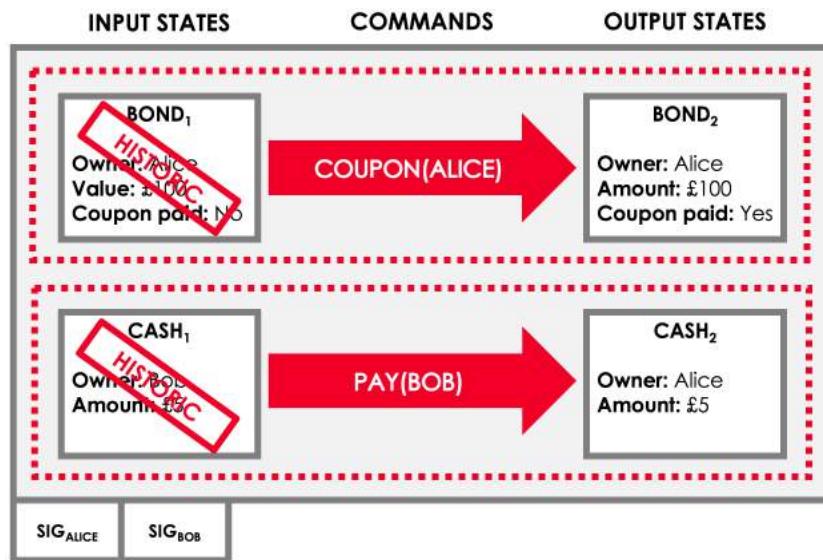
We can imagine that we'd want to impose different rules on what constitutes a valid transaction depending on whether this is a purchase or a coupon payment. For example, in the case of a purchase, we would require a change in the bond's current owner, whereas in the case of a coupon payment, we would 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:



## Attachments

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-window

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](#).

## Notary

A notary pool is a network service that provides uniqueness consensus by attesting that, for a given transaction, it has not already signed other transactions that consume any of the proposed transaction's input states. The notary pool provides the point of finality in the system.

Note that if the notary entity is absent then the transaction is not notarised at all. This is intended for issuance/genesis transactions that don't consume any other states and thus can't double spend anything. For more information on the notary services, see [Notaries](#).

## 7.5 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

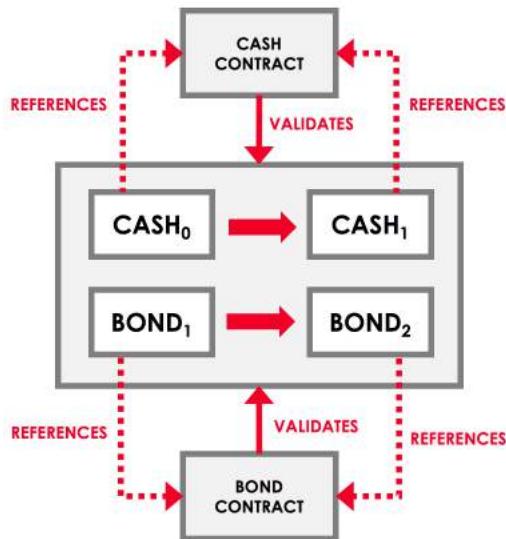
### 7.5.1 Transaction verification

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 transaction state specifies a *contract* type
- 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, time-window, 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.

### 7.5.2 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.

Future versions of Corda will evaluate transactions in a strictly 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.

Developers can pre-verify their CorDapps are deterministic by linking their CorDapps against the deterministic modules (see the [Deterministic Corda Modules](#)).

### 7.5.3 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.

### 7.5.4 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.

### 7.5.5 Legal prose

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.

## 7.6 Flows

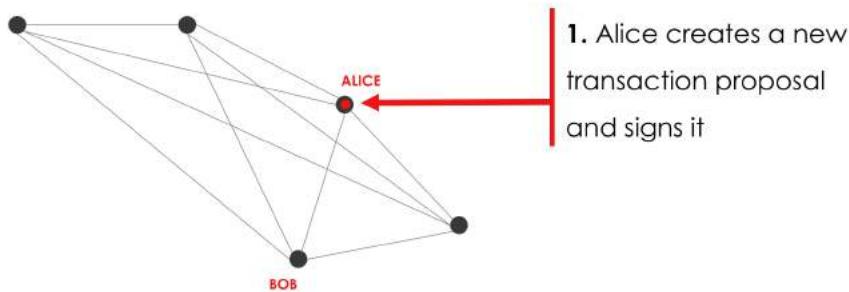
### Summary

- *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*

### 7.6.1 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.

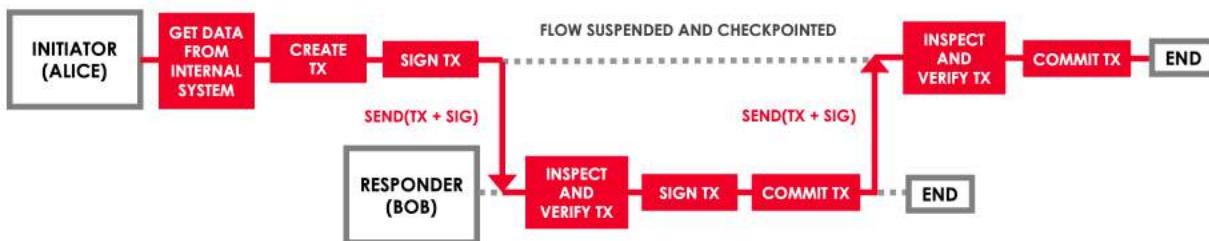
Here is a visualisation of the process of agreeing a simple ledger update between Alice and Bob:



### 7.6.2 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:



### 7.6.3 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 [API: Flows](#).

## 7.6.4 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.

## 7.7 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-spends

### 7.7.1 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

### 7.7.2 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:

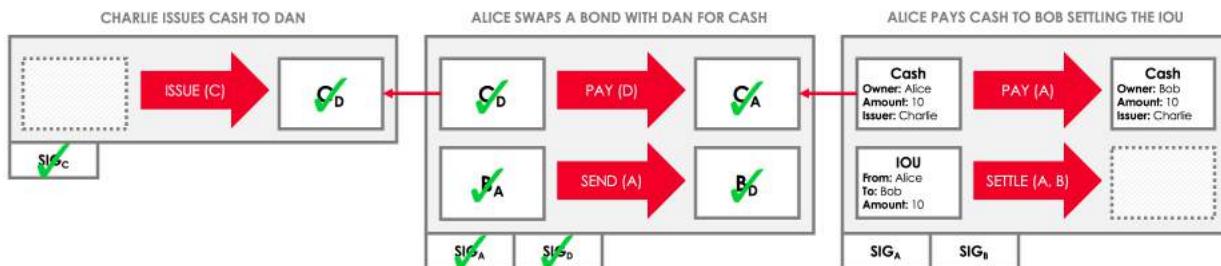
- The transaction is accepted by the contracts of every input and output state
- The transaction has all the required signatures

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:



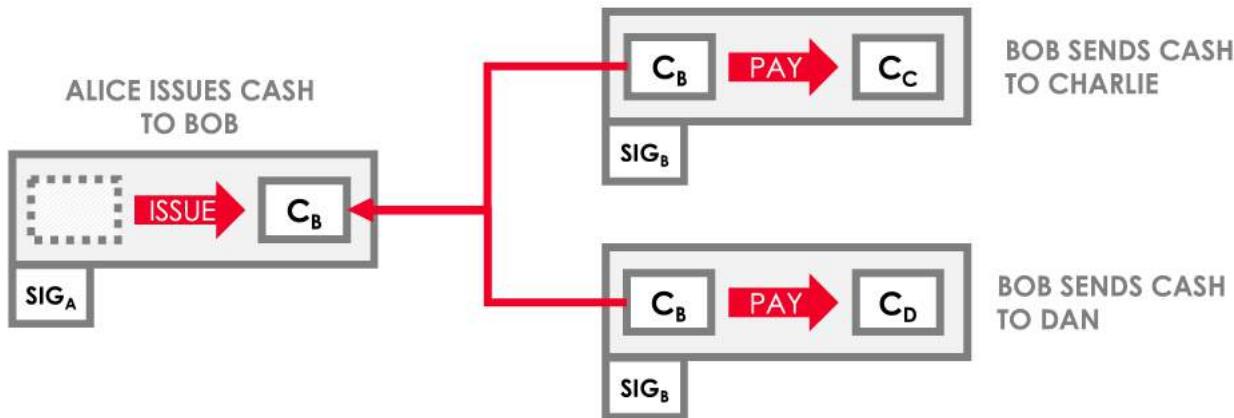
When verifying a proposed transaction, a given party may not have every transaction in the transaction chain that they need to verify. In this case, they can request the missing transactions from the transaction proposer(s). The transaction proposer(s) will always have the full transaction chain, since they would have requested it when verifying the transaction that created the transaction's input states.

### 7.7.3 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.

## 7.8 Notaries

### Summary

- Notary clusters prevent “double-spends”
- Notary clusters are also time-stamping authorities. If a transaction includes a time-window, it can only be notarised during that window
- Notary clusters may optionally also validate transactions, in which case they are called “validating” notaries, as opposed to “non-validating”
- A network can have several notary clusters, each running a different consensus algorithm

### 7.8.1 Overview

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

Upon being sent asked to notarise a transaction, a notary cluster 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 cluster provides the point of finality in the system. Until the notary cluster'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 cluster's signature is obtained, we can be sure that the proposed transaction's input states have not already been consumed by a prior transaction. Hence, notarisation is the point of finality in the system.

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

## 7.8.2 Consensus algorithms

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

In particular, notary clusters may differ in terms of:

- **Structure** - a notary cluster may be a single node, several mutually-trusting nodes, or several mutually-distrusting nodes
- **Consensus algorithm** - a notary cluster 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 cluster must also decide whether or not to provide **validity consensus** by validating each transaction before committing it. In making this decision, it faces the following trade-off:

- If a transaction **is not** checked for validity (non-validating notary), 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 cluster, causing the states to be marked as consumed
- If the transaction **is** checked for validity (validating notary), the notary will need to see the full contents of the transaction and its dependencies. This leaks potentially private data to the notary cluster

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 cluster can store 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 cluster sees.

### Data visibility

Below is a summary of what specific transaction components have to be revealed to each type of notary:

Transaction components	Validating	Non-validating
Input states	Fully visible	References only <sup>1</sup>
Output states	Fully visible	Hidden
Commands (with signer identities)	Fully visible	Hidden
Attachments	Fully visible	Hidden
Time window	Fully visible	Fully visible
Notary identity	Fully visible	Fully visible
Signatures	Fully visible	Hidden

Both types of notaries record the calling party's identity: the public key and the X.500 Distinguished Name.

### 7.8.3 Multiple notaries

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

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

### Changing notaries

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

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

Before these transactions can be created, the states must first all be re-pointed to the same notary cluster. 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 cluster has been changed

The input state's appointed notary cluster 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 cluster.

## 7.9 Vault

### 7.9.1 Soft Locking

Soft Locking is implemented in the vault to try and prevent a node constructing transactions that attempt to use the same input(s) simultaneously. Such transactions would result in naturally wasted work 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 outcome of such an eventuality will result in an `InsufficientBalanceException` for one of the requesters if there are insufficient number of fungible states available to satisfy both requests.

---

**Note:** The Cash Contract schema table is now automatically generated upon node startup as Coin Selection now uses this table to ensure correct locking and selection of states to satisfy minimum requested spending amounts.

---

<sup>1</sup> A state reference is composed of the issuing transaction's id and the state's position in the outputs. It does not reveal what kind of state it is or its contents.

Soft locks are also automatically applied within flows that issue or receive new states. These states are effectively soft locked until flow termination (exit or error) or by explicit release.

In addition, the `VaultService` exposes a number of functions a developer may use to explicitly reserve, release and query soft locks associated with states as required by their CordApp application logic:

```
/**
 * Reserve a set of [StateRef] for a given [UUID] unique identifier.
 * Typically, the unique identifier will refer to a [FlowLogic.runId]'s [UUID]
↳associated with an in-flight flow.
 * In this case if the flow terminates the locks will automatically be freed,
↳even if there is an error.
 * However, the user can specify their own [UUID] and manage this manually,
↳possibly across the lifetime of multiple
 * flows, or from other thread contexts e.g. [CordaService] instances.
 * In the case of coin selection, soft locks are automatically taken upon
↳gathering relevant unconsumed input refs.
 *
 * @throws [StatesNotAvailableException] when not possible to soft-lock all of
↳requested [StateRef].
 */
@Throws(StatesNotAvailableException::class)
fun softLockReserve(lockId: UUID, stateRefs: NonEmptySet<StateRef>)

/**
 * Release all or an explicitly specified set of [StateRef] for a given [UUID]
↳unique identifier.
 * A [Vault] soft-lock manager is automatically notified from flows that are
↳terminated, such that any soft locked
 * states may be released.
 * In the case of coin selection, soft-locks are automatically released once
↳previously gathered unconsumed
 * input refs are consumed as part of cash spending.
 */
fun softLockRelease(lockId: UUID, stateRefs: NonEmptySet<StateRef>? = null)
```

## Query

By default vault queries will always include locked states in its result sets. Custom filterable criteria can be specified using the `SoftLockingCondition` attribute of `VaultQueryCriteria`:

```
@CordaSerializable
data class SoftLockingCondition(val type: SoftLockingType, val lockIds: List<UUID>
↳= emptyList())

@CordaSerializable
enum class SoftLockingType {
    UNLOCKED_ONLY, // only unlocked states
    LOCKED_ONLY, // only soft locked states
    SPECIFIED, // only those soft locked states specified by lock id(s)
    UNLOCKED_AND_SPECIFIED // all unlocked states plus those soft locked states
↳specified by lock id(s)
}
```

## Explicit Usage

Soft locks are associated with transactions, and typically within the lifecycle of a flow. Specifically, every time a flow is started a soft lock identifier is associated with that flow for its duration (and released upon its natural termination or in the event of an exception). The `VaultSoftLockManager` is responsible within the Node for automatically managing this soft lock registration and release process for flows. The `TransactionBuilder` class has a new `lockId` field for the purpose of tracking lockable states. By default, it is automatically set to a random UUID (outside of a flow) or to a flow's unique ID (within a flow).

Upon building a new transaction to perform some action for a set of states on a contract, a developer must explicitly register any states they may wish to hold until that transaction is committed to the ledger. These states will be effectively ‘soft locked’ (not usable by any other transaction) until the developer explicitly releases these or the flow terminates or errors (at which point they are automatically released).

## Use Cases

A prime example where *soft locking* is automatically enabled is within the process of issuance and transfer of fungible state (eg. Cash). An issuer of some fungible asset (eg. Bank of Corda) may wish to transfer that new issue immediately to the issuance requester (eg. Big Corporation). This issuance and transfer operation must be *atomic* such that another flow (or instance of the same flow) does not step in and unintentionally spend the states issued by Bank of Corda before they are transferred to the intended recipient. Soft locking will automatically prevent new issued states within `IssuerFlow` from being spendable by any other flow until such time as the `IssuerFlow` itself terminates.

Other use cases for *soft locking* may involve competing flows attempting to match trades or any other concurrent activities that may involve operating on an identical set of unconsumed states.

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).

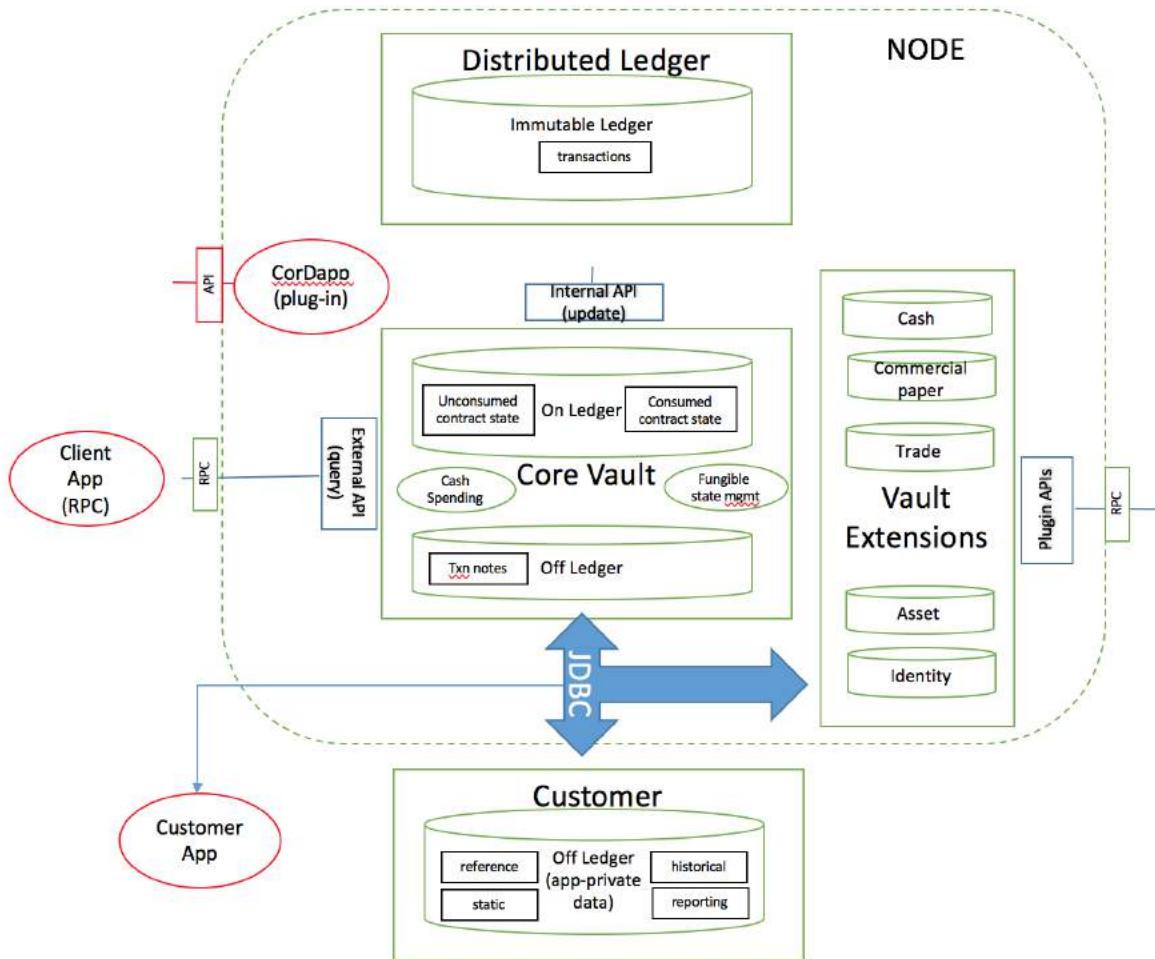
---

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.

## 7.10 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*

### 7.10.1 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.

### 7.10.2 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.

---

## 7.11 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

### 7.11.1 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.

For privacy purposes, the oracle does not require to have access on every part of the transaction and the only information it needs to see is their embedded, related to this oracle, command(s). We should also provide guarantees that all of the commands requiring a signature from this oracle should be visible to the oracle entity, but not the rest. To achieve that we use filtered transactions, in which the transaction proposer(s) uses a nested Merkle tree approach to "tear off" the unrelated parts of the transaction. See [Transaction tear-offs](#) for more information on how transaction tear-offs work.

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.

## 7.12 Nodes

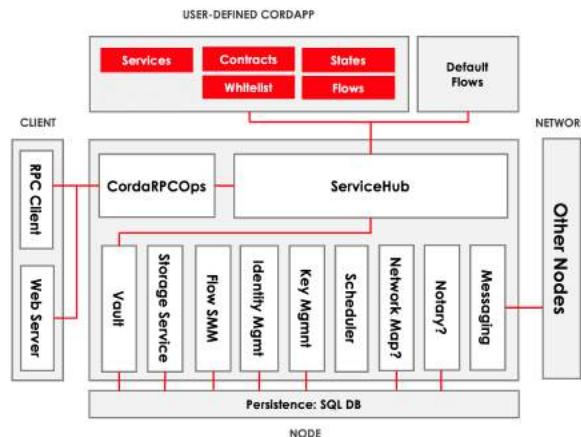
### Summary

- 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

### 7.12.1 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

### 7.12.2 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).

### 7.12.3 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.

### 7.12.4 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](#).

### 7.12.5 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

### 7.12.6 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

### 7.12.7 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 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.

The node can be safely shut down via a drain using the shell.

## 7.13 Transaction tear-offs

### Summary

- *Hide transaction components for privacy purposes*
- *Oracles and non-validating notaries can only see their “related” transaction components, but not the full transaction details*

### 7.13.1 Overview

There are cases where some of the entities involved on the transaction could only have partial visibility on the transaction parts. For instance, when an oracle should sign a transaction, the only information it needs to see is their embedded, related to this oracle, command(s). Similarly, a non-validating notary only needs to see a transaction's input states. Providing any additional transaction data to the oracle would constitute a privacy leak.

To combat this, we use the concept of filtered transactions, in which the transaction proposer(s) uses a nested Merkle tree approach 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.

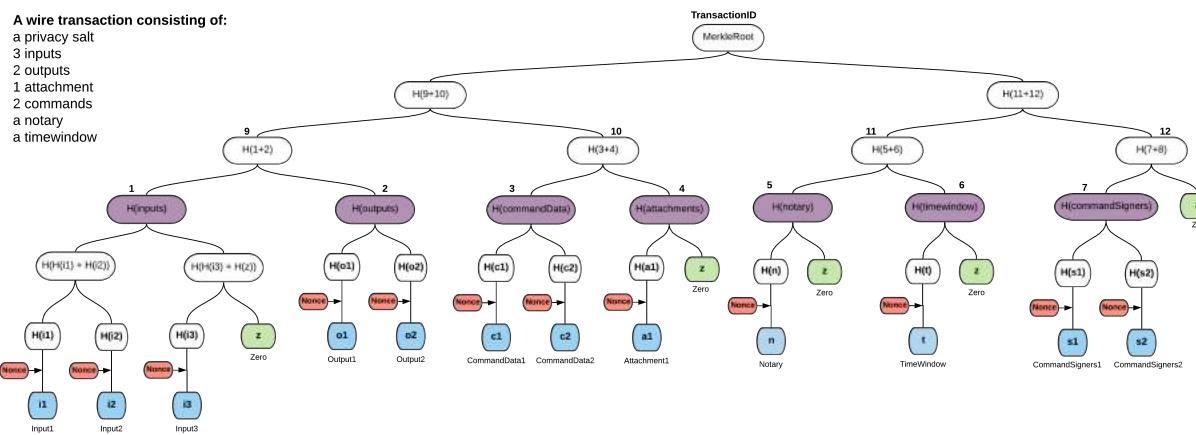
## Transaction Merkle trees

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 final nested tree structure also contains the other fields of the transaction, such as the time-window, the notary and the required signers. As shown in the picture below, the only component type that is requiring two trees instead of one is the command, which is split into command data and required signers for visibility purposes.

Corda is using a patent-pending approach using nested Merkle trees per component type. Briefly, a component subtree is generated for each component type (i.e., inputs, outputs, attachments). Then, the roots of these sub-trees form the leaves of the top Merkle tree and finally the root of this tree represents the transaction id.

Another important feature is that a nonce is deterministically generated for each component in a way that each nonce is independent. Then, we use the nonces along with their corresponding components to calculate the component hash, which is the actual Merkle tree leaf. Nonces are required to protect against brute force attacks that otherwise would reveal the content of low-entropy hashed values (i.e., a single-word text attachment).

After computing the leaves, each 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.

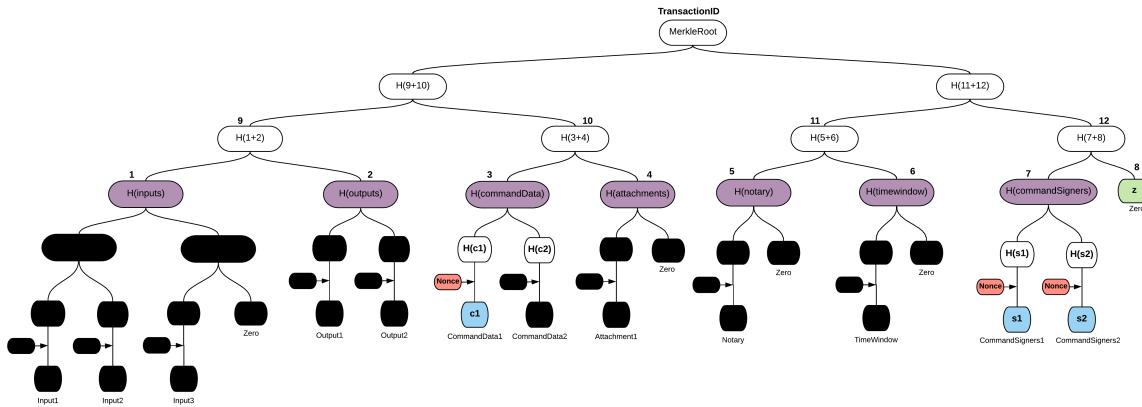


The transaction has three input states, two output states, two commands, one attachment, a notary and a time-window. Notice that if a tree is not a full binary tree, leaves are padded to the nearest power of 2 with zero hash (since finding a pre-image of sha256(x) == 0 is hard computational task) - marked 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 following we provide concrete examples on the data visible to an oracle and a non-validating notary, respectively.

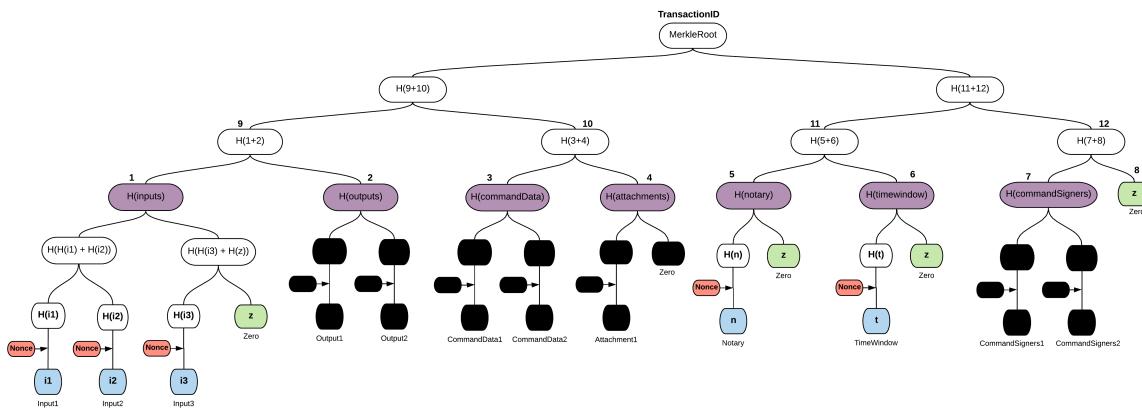
Let's assume that only the first command should be visible to an Oracle. We should also provide guarantees that all of the commands requiring a signature from this oracle should be visible to the oracle entity, but not the rest. Here is how this filtered transaction will be represented in the Merkle tree structure.



Blue nodes and H(c2) are provided to the Oracle service, while the black ones are omitted. H(c2) is required, so that the Oracle can compute H(commandData) without being able to see the second command, but at the same time ensuring CommandData1 is part of the transaction. It is highlighted that all signers are visible, so as to have a proof that no related command (that the Oracle should see) has been maliciously filtered out. Additionally, hashes of sub-trees (violet nodes) are also provided in the current Corda protocol. The latter is required for special cases, i.e., when required to know if a component group is empty or not.

Having all of the aforementioned data, one can calculate the root of the top tree and compare it with original transaction identifier - we have a proof that this command and time-window belong to this transaction.

Along the same lines, if we want to send the same transaction to a non-validating notary we should hide all components apart from input states, time-window and the notary information. This data is enough for the notary to know which input states should be checked for double-spending, if the time-window is valid and if this transaction should be notarised by this notary.



## 7.14 Trade-offs

## 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*

### 7.14.1 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

### 7.14.2 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 [Transaction tear-offs](#).
- **Key randomisation:** The parties to a transaction are identified only by their public keys, and fresh key pairs are generated for each transaction. As a result, an onlooker cannot identify which parties were involved in a given transaction.

### 7.14.3 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.

#### 7.14.4 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 should include a `LegalProseReference` link to the legal document describing the contract's intended behavior which can be relied upon to resolve conflicts

#### 7.14.5 Build vs. re-use

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

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

### 7.15 Deterministic JVM

#### Contents

- *Deterministic JVM*
  - *Introduction*
  - *Non-Determinism*
  - *Abstraction*
  - *Static Byte Code Analysis*
  - *Runtime Costing*
  - *Instrumentation and Rewriting*
  - *Future Work*
  - *Command-line Tool*

#### 7.15.1 Introduction

It is important that all nodes that process a transaction always agree on whether it is valid or not. Because transaction types are defined using JVM byte code, this means that the execution of that byte code must be fully deterministic. Out of the box a standard JVM is not fully deterministic, thus we must make some modifications in order to satisfy our requirements.

So, what does it mean for a piece of code to be fully deterministic? Ultimately, it means that the code, when viewed as a function, is pure. In other words, given the same set of inputs, it will always produce the same set of outputs without inflicting any side-effects that might later affect the computation.

---

**Important:** The code in the DJVM module has not yet been integrated with the rest of the platform. It will eventually become a part of the node and enforce deterministic and secure execution of smart contract code, which is mobile and may propagate around the network without human intervention.

Currently, it stands alone as an evaluation version. We want to give developers the ability to start trying it out and get used to developing deterministic code under the set of constraints that we envision will be placed on contract code in the future.

---

## 7.15.2 Non-Determinism

For a program running on the JVM, non-determinism could be introduced by a range of sources, for instance:

- **External input**, e.g., the file system, network, system properties and clocks.
- **Random number generators**.
- **Halting criteria**, e.g., different decisions about when to terminate long running programs.
- **Hash-codes**, or more specifically `Object.hashCode()`, which is typically implemented either by returning a pointer address or by assigning the object a random number. This could, for instance, surface as different iteration orders over hash maps and hash sets, or be used as non-pure input into arbitrary expressions.
- Differences in hardware **floating point arithmetic**.
- **Multi-threading** and consequent differences in scheduling strategies, affinity, etc.
- Differences in **API implementations** between nodes.
- **Garbage collector callbacks**.

To ensure that the contract verification function is fully pure even in the face of infinite loops we want to use a custom-built JVM sandbox. The sandbox performs static analysis of loaded byte code and a rewriting pass to allow for necessary instrumentation and constraint hardening.

The byte code rewriting further allows us to patch up and control the default behaviour of things like the hash-code generation for `java.lang.Object`. Contract code is rewritten the first time it needs to be executed and then stored for future use.

## 7.15.3 Abstraction

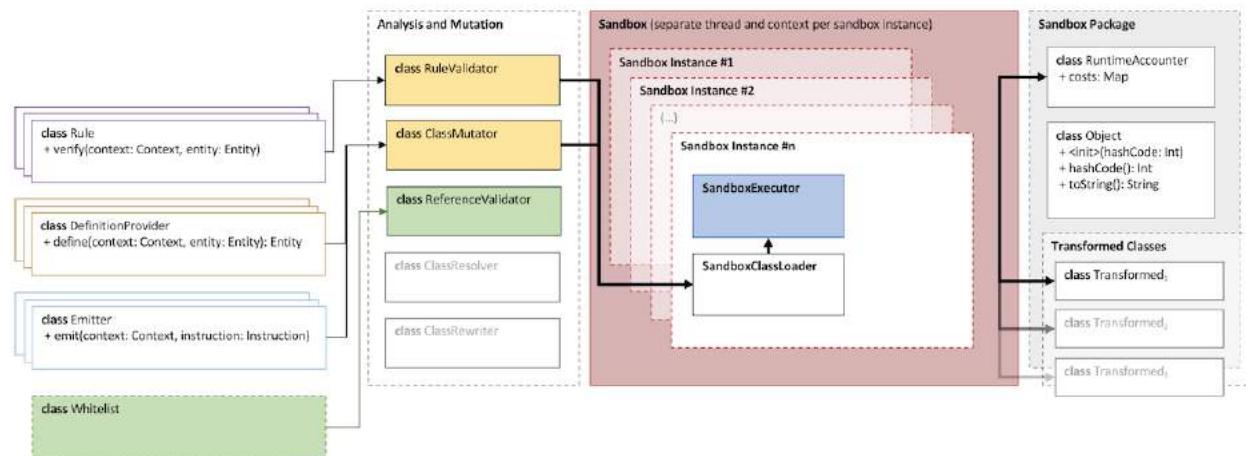
The sandbox is abstracted away as an executor which takes as input an implementation of the interface `Function<in Input, out Output>`, dereferenced by a `ClassSource`. This interface has a single method that needs implementing, namely `apply(Input): Output`.

A `ClassSource` object referencing such an implementation can be passed into the `SandboxExecutor<in Input, out Output>` together with an input of type `Input`. The executor has operations for both execution and static validation, namely `run()` and `validate()`. These methods both return a summary object.

- **In the case of execution, this summary object has information about:**
  - Whether or not the runnable was successfully executed.
  - If successful, the return value of `Function.apply()`.
  - If failed, the exception that was raised.

- And in both cases, a summary of all accrued costs during execution.
- **For validation, the summary contains:**
  - A type hierarchy of classes and interfaces loaded and touched by the sandbox’s class loader during analysis, each of which contain information about the respective transformations applied as well as meta-data about the types themselves and all references made from said classes.
  - A list of messages generated during the analysis. These can be of different severity, and only messages of severity `ERROR` will prevent execution.

The sandbox has a configuration that applies to the execution of a specific runnable. This configuration, on a higher level, contains a set of rules, definition providers and emitters.



The set of rules is what defines the constraints posed on the runtime environment. A rule can act on three different levels, namely on a type-, member- or instruction-level. The set of rules get processed and validated by the `RuleValidator` prior to execution.

Similarly, there is a set of definition providers which can be used to modify the definition of either a type or a type’s members. This is what controls things like ensuring that all methods implement strict floating point arithmetic, and normalisation of synchronised methods.

Lastly, there is a set of emitters. These are used to instrument the byte code for cost accounting purposes, and also to inject code for checks that we want to perform at runtime or modifications to out-of-the-box behaviour. Many of these emitters will rewrite non-deterministic operations to throw `RuleViolationError` exceptions instead, which means that the ultimate proof that a function is *truly* deterministic is that it executes successfully inside the DJVM.

#### 7.15.4 Static Byte Code Analysis

In summary, the byte code analysis currently performs the following checks. This is not an exhaustive list as further work may well introduce additional constraints that we would want to place on the sandbox environment.

- *Disallow Catching ThreadDeath Exception*
- *Disallow Catching ThresholdViolationException*
- *Disallow Dynamic Invocation*
- *Disallow Native Methods*
- *Disallow Finalizer Methods*

- *Disallow Overridden Sandbox Package*
- *Disallow Breakpoints*
- *Disallow Reflection*
- *Disallow Unsupported API Versions*

---

**Note:** It is worth noting that not only smart contract code is instrumented by the sandbox, but all code that it can transitively reach. In particular this means that the Java runtime classes and any other library code used in the program are also instrumented and persisted ahead of time.

---

### Disallow Catching ThreadDeath Exception

Prevents exception handlers from catching `ThreadDeath` exceptions. If the developer attempts to catch an `Error` or a `Throwable` (both being transitive parent types of `ThreadDeath`), an explicit check will be injected into the byte code to verify that exceptions that are trying to kill the current thread are not being silenced. Consequently, the user will not be able to bypass an exit signal.

### Disallow Catching ThresholdViolationException

The `ThresholdViolationException` is, as the name suggests, used to signal to the sandbox that a cost tracked by the runtime cost accountant has been breached. For obvious reasons, the sandbox needs to protect against user code that tries to catch such exceptions, as doing so would allow the user to bypass the thresholds set out in the execution profile.

### Disallow Dynamic Invocation

Forbids `invokedynamic` byte code as the libraries that support this functionality have historically had security problems and it is primarily needed only by scripting languages. In the future, this constraint will be eased to allow for dynamic invocation in the specific lambda and string concatenation meta-factories used by Java code itself.

### Disallow Native Methods

Forbids native methods as these provide the user access into operating system functionality such as file handling, network requests, general hardware interaction, threading, *etc.* These all constitute sources of non-determinism, and allowing such code to be called arbitrarily from the JVM would require deterministic guarantees on the native machine code level. This falls out of scope for the DJVM.

### Disallow Finalizer Methods

Forbids finalizers as these can be called at unpredictable times during execution, given that their invocation is controlled by the garbage collector. As stated in the standard Java documentation:

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

### **Disallow Overridden Sandbox Package**

Forbids attempts to override rewritten classes. For instance, loading a class `com.foo.Bar` into the sandbox, analyses it, rewrites it and places it into `sandbox.com.foo.Bar`. Attempts to place originating classes in the top-level sandbox package will therefore fail as this poses a security risk. Doing so would essentially bypass rule validation and instrumentation.

### **Disallow Breakpoints**

For obvious reasons, the breakpoint operation code is forbidden as this can be exploited to unpredictably suspend code execution and consequently interfere with any time bounds placed on the execution.

### **Disallow Reflection**

For now, the use of reflection APIs is forbidden as the unmanaged use of these can provide means of breaking out of the protected sandbox environment.

### **Disallow Unsupported API Versions**

Ensures that loaded classes are targeting an API version between 1.5 and 1.8 (inclusive). This is merely to limit the breadth of APIs from the standard runtime that needs auditing.

## **7.15.5 Runtime Costing**

The runtime accountant inserts calls to an accounting object before expensive byte code. The goal of this rewrite is to deterministically terminate code that has run for an unacceptably long amount of time or used an unacceptable amount of memory. Types of expensive byte code include method invocation, memory allocation, branching and exception throwing.

The cost instrumentation strategy used is a simple one: just counting byte code that are known to be expensive to execute. The methods can be limited in size and jumps count towards the costing budget, allowing us to determine a consistent halting criteria. However it is still possible to construct byte code sequences by hand that take excessive amounts of time to execute. The cost instrumentation is designed to ensure that infinite loops are terminated and that if the cost of verifying a transaction becomes unexpectedly large (*e.g.*, contains algorithms with complexity exponential in transaction size) that all nodes agree precisely on when to quit. It is not intended as a protection against denial of service attacks. If a node is sending you transactions that appear designed to simply waste your CPU time then simply blocking that node is sufficient to solve the problem, given the lack of global broadcast.

The budgets are separate per operation code type, so there is no unified cost model. Additionally the instrumentation is high overhead. A more sophisticated design would be to calculate byte code costs statically as much as possible ahead of time, by instrumenting only the entry point of ‘accounting blocks’, *i.e.*, runs of basic blocks that end with either a method return or a backwards jump. Because only an abstract cost matters (this is not a profiler tool) and because the limits are expected to be set relatively high, there is no need to instrument every basic block. Using the max of both sides of a branch is sufficient when neither branch target contains a backwards jump. This sort of design will be investigated if the per category budget accounting turns out to be insufficient.

A further complexity comes from the need to constrain memory usage. The sandbox imposes a quota on bytes allocated rather than bytes retained in order to simplify the implementation. This strategy is unnecessarily harsh on smart contracts that churn large quantities of garbage yet have relatively small peak heap sizes and, again, it may be that in practice a more sophisticated strategy that integrates with the garbage collector is required in order to set quotas to a usefully generic level.

---

**Note:** The current thresholds have been set arbitrarily for demonstration purposes and should not be relied upon as sensible defaults in a production environment.

---

## 7.15.6 Instrumentation and Rewriting

### Always Use Strict Floating Point Arithmetic

Sets the `strictfp` flag on all methods, which requires the JVM to do floating point arithmetic in a hardware independent fashion. Whilst we anticipate that floating point arithmetic is unlikely to feature in most smart contracts (big integer and big decimal libraries are available), it is available for those who want to use it.

### Always Use Exact Math

Replaces integer and long addition and multiplication with calls to `Math.addExact()` and `Math.multiplyExact`, respectively. Further work can be done to implement exact operations for increments, decrements and subtractions as well. These calls into `java.lang.Math` essentially implement checked arithmetic over integers, which will throw an exception if the operation overflows.

### Always Inherit From Sandboxed Object

As mentioned further up, `Object.hashCode()` is typically implemented using either the memory address of the object or a random number; which are both non-deterministic. The DJVM shields the runtime from this source of non-determinism by rewriting all classes that inherit from `java.lang.Object` to derive from `sandbox.java.lang.Object` instead. This sandboxed `Object` implementation takes a hash-code as an input argument to the primary constructor, persists it and returns the value from the `hashCode()` method implementation. It also has an overridden implementation of `toString()`.

The loaded classes are further rewritten in two ways:

- All allocations of new objects of type `java.lang.Object` get mapped into using the sandboxed object.
- Calls to the constructor of `java.lang.Object` get mapped to the constructor of `sandbox.java.lang.Object` instead, passing in a constant value for now. In the future, we can easily have this passed-in hash-code be a pseudo random number seeded with, for instance, the hash of the transaction or some other dynamic value, provided of course that it is deterministically derived.

### Disable Synchronised Methods and Blocks

The DJVM doesn't support multi-threading and so synchronised methods and code blocks have little use in sandboxed code. Consequently, we automatically transform them into ordinary methods and code blocks instead.

## 7.15.7 Future Work

Further work is planned:

- To enable controlled use of reflection APIs.
- Currently, dynamic invocation is disallowed. Allow specific lambda and string concatenation meta-factories used by Java code itself.
- Map more mathematical operations to use their 'exact' counterparts.

- General tightening of the enforced constraints.
- Cost accounting of runtime metrics such as memory allocation, branching and exception handling. More specifically defining sensible runtime thresholds and make further improvements to the instrumentation.
- More sophisticated runtime accounting as discussed in [Runtime Costing](#).

### 7.15.8 Command-line Tool

Open your terminal and navigate to the `djvm` directory in the Corda source tree. Then issue the following command:

```
$ djvm/shell/install
```

This will build the DJVM tool and install a shortcut on Bash-enabled systems. It will also generate a Bash completion file and store it in the `shell` folder. This file can be sourced from your Bash initialisation script.

```
$ cd ~  
$ djvm
```

Now, you can create a new Java file from a skeleton that `djvm` provides, compile the file, and consequently run it by issuing the following commands:

```
$ djvm new Hello  
$ vim tmp/net/corda/sandbox/Hello.java  
$ djvm build Hello  
$ djvm run Hello
```

This run will produce some output similar to this:

```
Running class net.corda.sandbox.Hello...  
Execution successful  
- result = null  
  
Runtime Cost Summary:  
- allocations = 0  
- invocations = 1  
- jumps = 0  
- throws = 0
```

The output should be pretty self-explanatory, but just to summarise:

- It prints out the return value from the `Function<Object, Object>.apply()` method implemented in `net.corda.sandbox.Hello`.
- It also prints out the aggregated costs for allocations, invocations, jumps and throws.

Other commands to be aware of are:

- `djvm check` which allows you to perform some up-front static analysis without running the code. However, be aware that the DJVM also transforms some non-deterministic operations into `RuleViolationError` exceptions. A successful `check` therefore does *not* guarantee that the code will behave correctly at runtime.
- `djvm inspect` which allows you to inspect what byte code modifications will be applied to a class.
- `djvm show` which displays the transformed byte code of a class, *i.e.*, the end result and not the difference.

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

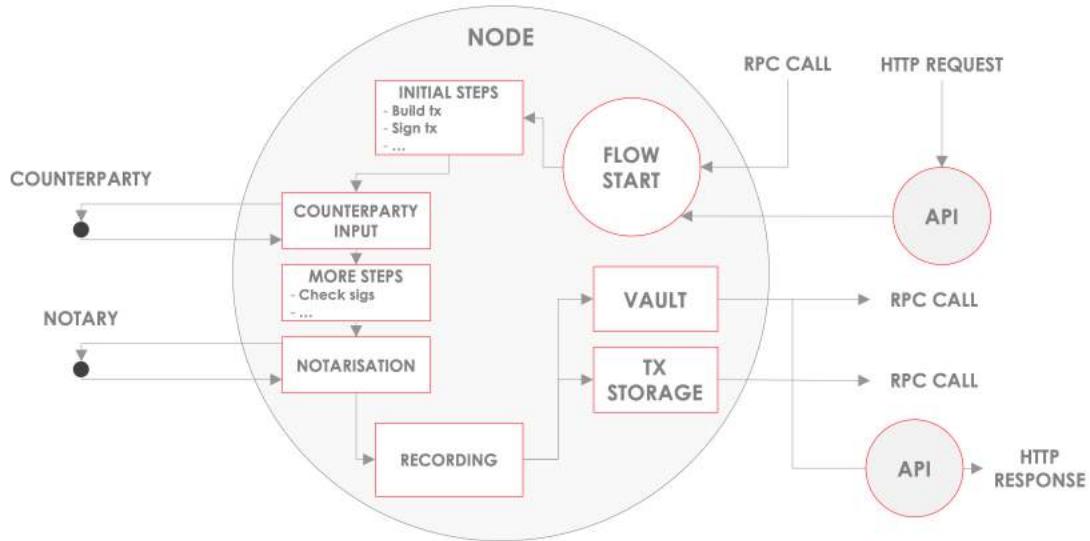
- [Corda: An Introduction](#)
- [Corda: A Distributed Ledger \(A.K.A. the Technical White Paper\)](#)

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

## CORDAPPS

### 8.1 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 over RPC:



#### 8.1.1 CorDapp components

CorDapps take the form of a set of JAR files containing class definitions written in Java and/or Kotlin.

These class definitions will commonly include the following elements:

- Flows: Define a routine for the node to run, usually to update the ledger (see [Key Concepts - Flows](#)). They subclass FlowLogic
- States: Define the facts over which agreement is reached (see [Key Concepts - States](#)). They implement the ContractState interface
- Contracts, defining what constitutes a valid ledger update (see [Key Concepts - Contracts](#)). They implement the Contract interface
- Services, providing long-lived utilities within the node. They subclass SingletonSerializationToken

- Serialisation whitelists, restricting what types your node will receive off the wire. They implement the `SerializationWhitelist` interface

But the CorDapp JAR can also include other class definitions. These may 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
- Utility classes

### 8.1.2 An example

Suppose a node owner wants their node to be able to trade bonds. They may choose to install a 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.

### 8.1.3 Writing and building apps that run on both Corda (open source) and Corda Enterprise

Corda and Corda Enterprise are compatible and interoperable, which means you can write a CorDapp that can run on both. To make this work in practice you should follow these steps:

1. Ensure your CorDapp is designed per [Structuring a CorDapp](#) and annotated according to [CorDapp separation](#). In particular, it is critical to separate the consensus-critical parts of your application (contracts, states and their dependencies) from the rest of the business logic (flows, APIs, etc). The former - the **CorDapp kernel** - is the Jar that will be attached to transactions creating/consuming your states and is the Jar that any node on the network verifying the transaction must execute.

---

**Note:** It is also important to understand how to manage any dependencies a CorDapp may have on 3rd party libraries and other CorDapps. Please read [Setting your dependencies](#) to understand the options and recommendations with regards to correctly Jar'ing CorDapp dependencies.

---

2. Compile this **CorDapp kernel** Jar once, and then depend on it from your workflows Jar (or Jars - see below). Importantly, if you want your app to work on both Corda and Corda Enterprise, you must compile this Jar against Corda, not Corda Enterprise. This is because, in future, we may add additional functionality to Corda Enterprise that is not in Corda and you may inadvertently create a CorDapp kernel that does not work on Corda open source. Compiling against Corda open source as a matter of course prevents this risk, as well as preventing the risk that you inadvertently create two different versions of the Jar, which will have different hashes and hence break compatibility and interoperability.

---

**Note:** As of Corda 4 it is recommended to use [CorDapp Jar signing](#) to leverage the new signature constraints functionality.

---

3. Your workflow Jar(s) should depend on the **CorDapp kernel** (contract, states and dependencies). Importantly, you can create different workflow Jars for Corda and Corda Enterprise, because the workflows Jar is not consensus critical. For example, you may wish to add additional features to your CorDapp for when it is run on Corda Enterprise (perhaps it uses advanced features of one of the supported enterprise databases or includes advanced database migration scripts, or some other Enterprise-only feature).

In summary, structure your app as kernel (contracts, states, dependencies) and workflow (the rest) and be sure to compile the kernel against Corda open source. You can compile your workflow (Jars) against the distribution of Corda that they target.

## 8.2 Getting set up for CorDapp development

### 8.2.1 Software requirements

Corda uses industry-standard tools:

- **Java 8 JVM** - we require at least version 8u171, but do not currently support Java 9 or higher.

We have tested with the following builds:

- Oracle JDK
- Amazon Corretto
- Red Hat's OpenJDK
- Zulu's OpenJDK

Please note that OpenJDK builds usually exclude JavaFX, which our GUI tools require.

- **IntelliJ IDEA** - supported versions **2017.x**, **2018.x** and **2019.x** (with Kotlin plugin version 1.2.71)
- **Gradle** - we use 4.10 and the `gradlew` script in the project / samples directories will download it for you.

Please note:

- Applications on Corda (CorDapps) can be written in any language targeting the JVM. However, Corda itself and most of the samples are written in Kotlin. Kotlin is an [official Android language](#), and you can read more about why Kotlin is a strong successor to Java [here](#). If you're unfamiliar with Kotlin, there is an official [getting started guide](#), and a series of [Kotlin Koans](#)
- IntelliJ IDEA is recommended due to the strength of its Kotlin integration.

Following these software recommendations will minimize the number of errors you encounter, and make it easier for others to provide support. However, if you do use other tools, we'd be interested to hear about any issues that arise.

### 8.2.2 Set-up instructions

The instructions below will allow you to set up your development environment for running Corda and writing CorDapps. If you have any issues, please reach out on [Stack Overflow](#) or via [our Slack channels](#).

The set-up instructions are available for the following platforms:

- windows-label (or in video form)

- *Mac* (or in video form)
- *Debian/Ubuntu*
- *Fedora*

---

**Note:** These setup instructions will guide you on how to install the Oracle JDK. Each JDK can be found on their respective sites:

- Oracle
  - Amazon Corretto
  - Red Hat OpenJDK
  - Zulu OpenJDK
- 

### 8.2.3 Windows

**Warning:** If you are using a Mac, Debian/Ubuntu or Fedora machine, please follow the *Mac*, *Debian/Ubuntu* or *Fedora* instructions instead.

#### Java

1. Visit <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
2. Click the download link for jdk-8uXXX-windows-x64.exe (where “XXX” is the latest minor version number)
3. Download and run the executable to install Java (use the default settings)
4. Add Java to the PATH environment variable by following the instructions in the [Oracle documentation](#)
5. Open a new command prompt and run `java -version` to test that Java is installed correctly

#### Git

1. Visit <https://git-scm.com/download/win>
2. Click the “64-bit Git for Windows Setup” download link.
3. Download and run the executable to install Git (use the default settings)
4. Open a new command prompt and type `git --version` to test that git is installed correctly

#### IntelliJ

1. Visit <https://www.jetbrains.com/idea/download/download-thanks.html?code=IIC>
2. Download and run the executable to install IntelliJ Community Edition (use the default settings)
3. Ensure the Kotlin plugin in IntelliJ is updated to version 1.2.71 (new installs will contain this version)

## 8.2.4 Mac

**Warning:** If you are using a Windows, Debian/Ubuntu or Fedora machine, please follow the windows-label, [Debian/Ubuntu](#) or [Fedora](#) instructions instead.

### Java

1. Visit <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
2. Click the download link for jdk-8uXXX-macosx-x64.dmg (where “XXX” is the latest minor version number)
3. Download and run the executable to install Java (use the default settings)
4. Open a new terminal window and run `java -version` to test that Java is installed correctly

### IntelliJ

1. Visit <https://www.jetbrains.com/idea/download/download-thanks.html?platform=mac&code=IIC>
2. Download and run the executable to install IntelliJ Community Edition (use the default settings)
3. Ensure the Kotlin plugin in IntelliJ is updated to version 1.2.71 (new installs will contain this version)

## 8.2.5 Debian/Ubuntu

**Warning:** If you are using a Mac, Windows or Fedora machine, please follow the [Mac](#), windows-label or [Fedora](#) instructions instead.

These instructions were tested on Ubuntu Desktop 18.04 LTS.

### Java

1. Open a new terminal and add the Oracle PPA to your repositories by typing `sudo add-apt-repository ppa:webupd8team/java`. Press ENTER when prompted.
2. Update your packages list with the command `sudo apt update`
3. Install the Oracle JDK 8 by typing `sudo apt install oracle-java8-installer`. Press Y when prompted and agree to the licence terms.
4. Verify that the JDK was installed correctly by running `java -version`

### Git

1. From the terminal, Git can be installed using apt with the command `sudo apt install git`
2. Verify that git was installed correctly by typing `git --version`

## IntelliJ

Jetbrains offers a pre-built snap package that allows for easy, one-step installation of IntelliJ onto Ubuntu.

1. To download the snap, navigate to <https://snapcraft.io/intellij-idea-community>
2. Click Install, then View in Desktop Store. Choose Ubuntu Software in the Launch Application window.
3. Ensure the Kotlin plugin in IntelliJ is updated to version 1.2.71 (new installs will contain this version)

## 8.2.6 Fedora

**Warning:** If you are using a Mac, Windows or Debian/Ubuntu machine, please follow the [Mac](#), [windows-label](#) or [Debian/Ubuntu](#) instructions instead.

These instructions were tested on Fedora 28.

### Java

1. Download the RPM installation file of Oracle JDK from <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.
2. Install the package with `rpm -ivh jdk-<version>-linux-<architecture>.rpm` or use the default software manager.
3. Choose java version by using the following command `alternatives --config java`
4. Verify that the JDK was installed correctly by running `java -version`

### Git

1. From the terminal, Git can be installed using dnf with the command `sudo dnf install git`
2. Verify that git was installed correctly by typing `git --version`

## IntelliJ

1. Visit <https://www.jetbrains.com/idea/download/download-thanks.html?platform=linux&code=IIC>
2. Unpack the `.tar.gz` file using the following command `tar xfz ideaIC-<version>.tar.gz -C /opt`
3. Run IntelliJ with `/opt/ideaIC-<version>/bin/idea.sh`
4. Ensure the Kotlin plugin in IntelliJ is updated to version 1.2.71 (new installs will contain this version)

## 8.2.7 Next steps

First, run the *example CorDapp*.

Next, read through the *Corda Key Concepts* to understand how Corda works.

By then, you'll be ready to start writing your own CorDapps. Learn how to do this in the [Hello, World tutorial](#). You may want to refer to the [API documentation](#), the [flow cookbook](#) and the [samples](#) along the way.

If you encounter any issues, please ask on [Stack Overflow](#) or via our Slack channels.

## 8.3 Running the example CorDapp

### Contents

- [\*Running the example CorDapp\*](#)
  - [\*Downloading the example CorDapp\*](#)
  - [\*Opening the example CorDapp in IntelliJ\*](#)
    - \* [\*Project structure\*](#)
  - [\*Running the example CorDapp\*](#)
    - \* [\*Running the example CorDapp from the terminal\*](#)
      - [\*Building the example CorDapp\*](#)
      - [\*Running the example CorDapp\*](#)
    - \* [\*Running the example CorDapp from IntelliJ\*](#)
  - [\*Interacting with the example CorDapp\*](#)
    - \* [\*Via HTTP\*](#)
      - [\*Creating an IOU via the endpoint\*](#)
      - [\*Submitting an IOU via the web front-end\*](#)
      - [\*Checking the output\*](#)
    - \* [\*Via the interactive shell \(terminal only\)\*](#)
      - [\*Creating an IOU via the interactive shell\*](#)
      - [\*Checking the output\*](#)
    - \* [\*Via the h2 web console\*](#)
  - [\*Running nodes across machines\*](#)
  - [\*Testing your CorDapp\*](#)
    - \* [\*Contract tests\*](#)
    - \* [\*Flow tests\*](#)
    - \* [\*Integration tests\*](#)
    - \* [\*Running tests in IntelliJ\*](#)
  - [\*Debugging your CorDapp\*](#)

The example CorDapp allows nodes to agree IOUs with each other, as long as they obey the following contract rules:

- The IOU's value is strictly positive
- A node is not trying to issue an IOU to itself

We will deploy and run the CorDapp on four test nodes:

- **Notary**, which runs a notary service
- **PartyA**
- **PartyB**
- **PartyC**

Because data is only propagated on a need-to-know basis, any IOUs agreed between PartyA and PartyB become “shared facts” between PartyA and PartyB only. PartyC won’t be aware of these IOUs.

### 8.3.1 Downloading the example CorDapp

Start by downloading the example CorDapp from GitHub:

- Set up your machine by following the [quickstart guide](#)
- Clone the samples repository from using the following command: `git clone https://github.com/corda/samples`
- Change directories to the `cordapp-example` folder: `cd samples/cordapp-example`

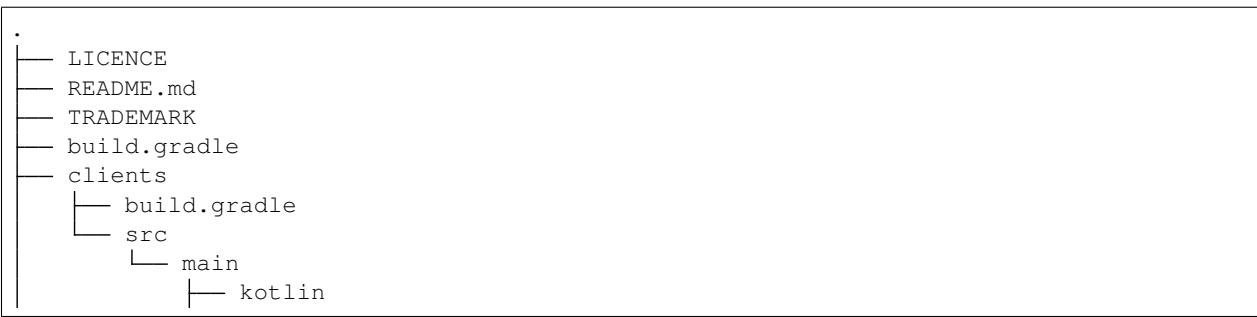
### 8.3.2 Opening the example CorDapp in IntelliJ

Let’s open the example CorDapp in IntelliJ IDEA:

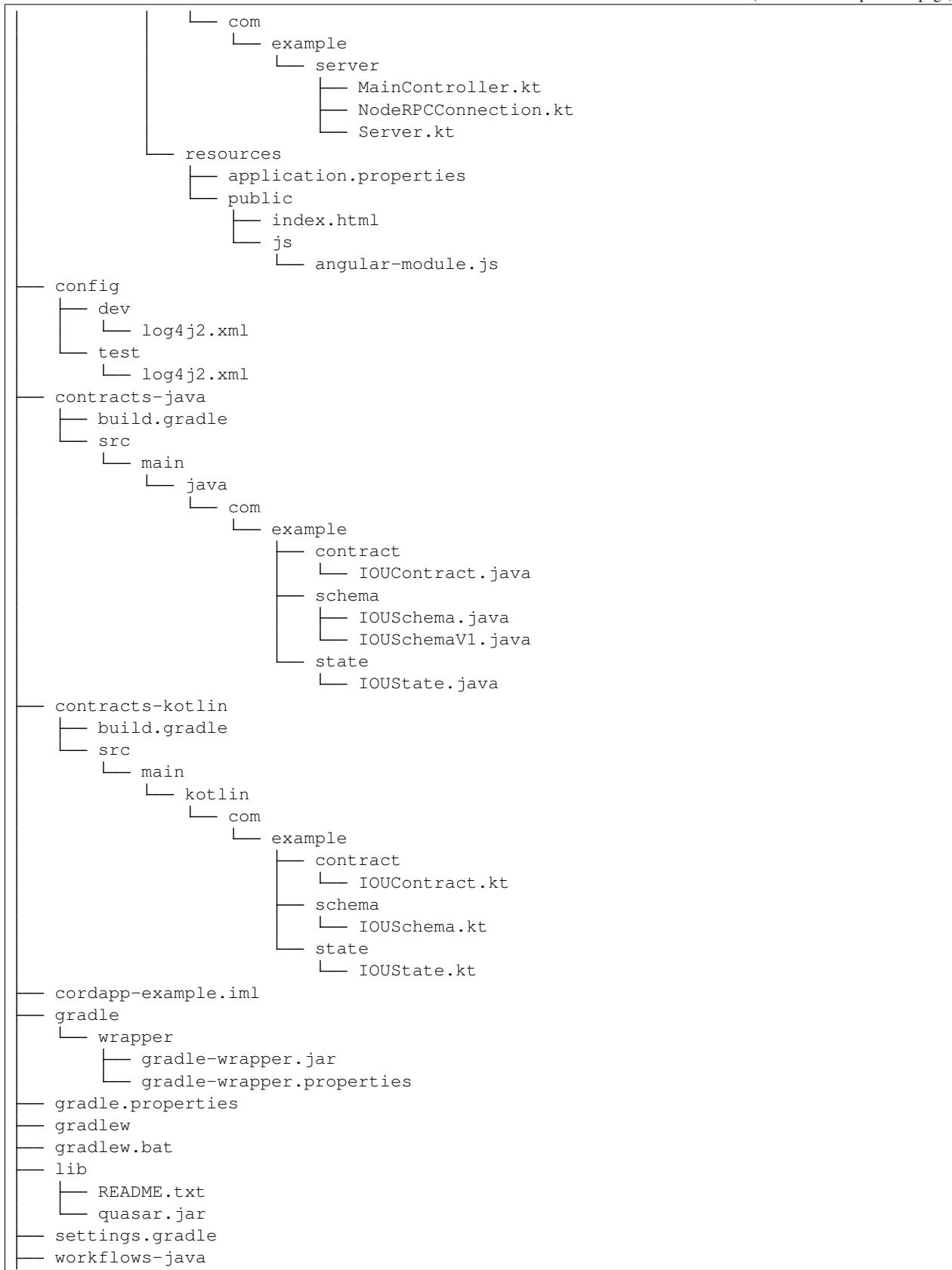
- Open IntelliJ
- A splash screen will appear. Click open, navigate to and select the `cordapp-example` folder, and click OK
- Once the project is open, click File, then Project Structure. Under Project SDK:, set the project SDK by clicking New..., clicking JDK, and navigating to C:\Program Files\Java\jdk1.8.0\_XXX on Windows or Library/Java/JavaVirtualMachines/jdk1.8.XXX on MacOSX (where XXX is the latest minor version number). Click Apply followed by OK
- Again under File then Project Structure, select Modules. Click +, then Import Module, then select the `cordapp-example` folder and click Open. Choose to Import module from external model, select Gradle, click Next then Finish (leaving the defaults) and OK
- Gradle will now download all the project dependencies and perform some indexing. This usually takes a minute or so

#### Project structure

The example CorDapp has the following structure:

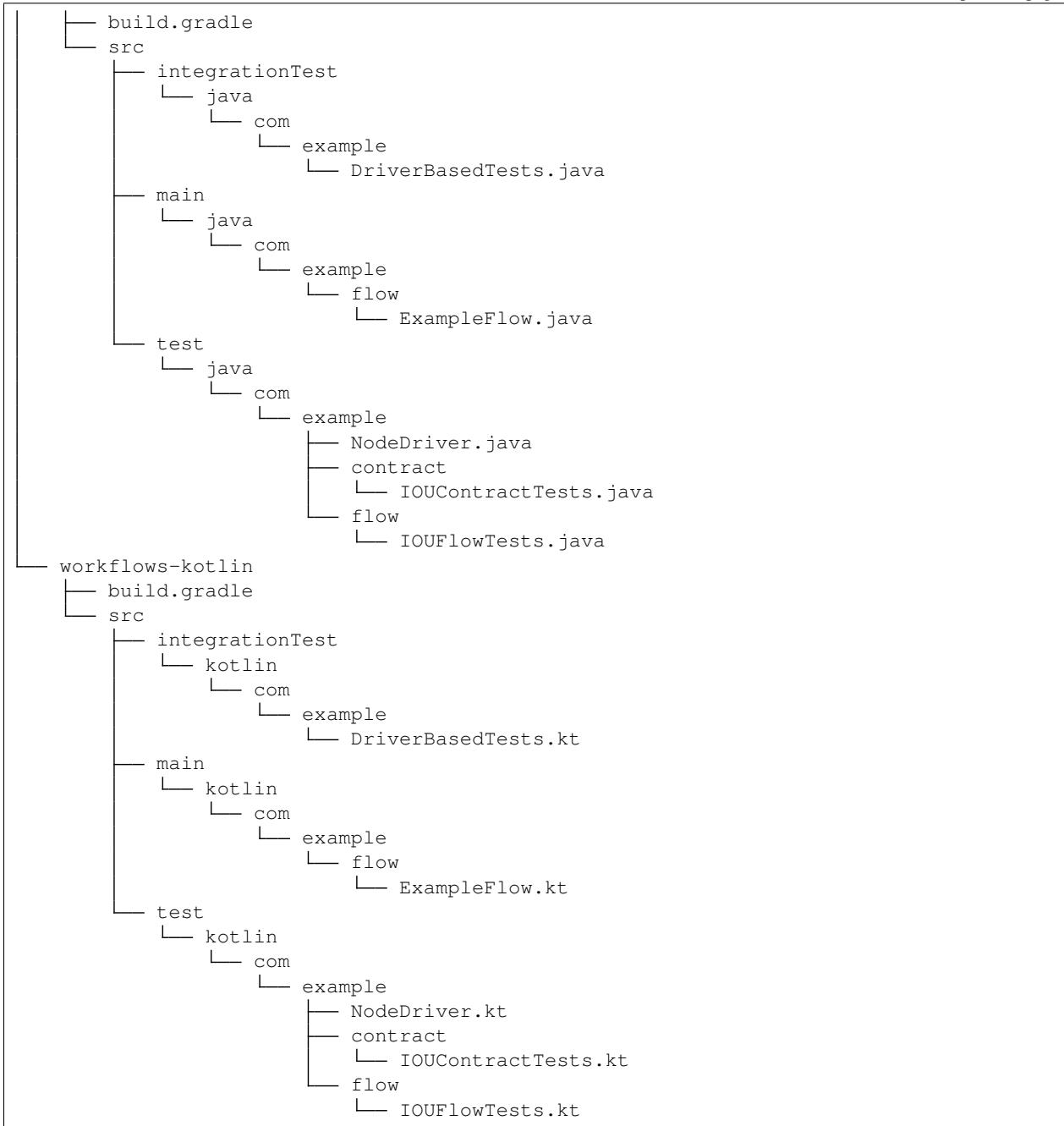


(continued from previous page)



(continues on next page)

(continued from previous page)



The key files and directories are as follows:

- The **root directory** contains some gradle files, a README and a LICENSE
- **config** contains log4j2 configs
- **gradle** contains the gradle wrapper, which allows the use of Gradle without installing it yourself and worrying about which version is required
- **lib** contains the Quasar jar which rewrites our CorDapp's flows to be checkpointable
- **clients** contains the source code for spring boot integration

- **contracts-java** and **workflows-java** contain the source code for the example CorDapp written in Java
- **contracts-kotlin** and **workflows-kotlin** contain the same source code, but written in Kotlin. CorDapps can be developed in either Java and Kotlin

### 8.3.3 Running the example CorDapp

There are two ways to run the example CorDapp:

- Via the terminal
- Via IntelliJ

Both approaches will create a set of test nodes, install the CorDapp on these nodes, and then run the nodes. You can read more about how we generate nodes [here](#).

#### Running the example CorDapp from the terminal

#### Building the example CorDapp

- Open a terminal window in the `cordapp-example` directory
- Run the `deployNodes` Gradle task to build four nodes with our CorDapp already installed on them:
  - Unix/Mac OSX: `./gradlew deployNodes`
  - Windows: `gradlew.bat deployNodes`

---

**Note:** CorDapps can be written in any language targeting the JVM. In our case, we've provided the example source in both Kotlin and Java. Since both sets of source files are functionally identical, we will refer to the Kotlin version throughout the documentation.

---

- After the build finishes, you will see the following output in the `workflows-kotlin/build/nodes` folder:
  - A folder for each generated node
  - A `runnodes` shell script for running all the nodes simultaneously on osX
  - A `runnodes.bat` batch file for running all the nodes simultaneously on Windows
- Each node in the `nodes` folder will have the following structure:

```
. nodeName
└── additional-node-infos  //
└── certificates
└── corda.jar              // The Corda node runtime
└── cordapps                // The node's CorDapps
    └── corda-finance-contracts-4.1.jar
    └── corda-finance-workflows-4.1.jar
    └── cordapp-example-0.1.jar
└── drivers
└── logs
└── network-parameters
└── node.conf                // The node's configuration file
└── nodeInfo-<HASH>          // The hash will be different each time you generate a_
    └── node
        └── persistence.mv.db   // The node's database
```

**Note:** `deployNodes` is a utility task to create an entirely new set of nodes for testing your CorDapp. In production, you would instead create a single node as described in [Creating nodes locally](#) and build your CorDapp JARs as described in [Building and installing a CorDapp](#).

## Running the example CorDapp

Start the nodes by running the following command from the root of the `cordapp-example` folder:

- Unix/Mac OSX: workflows-kotlin/build/nodes/runnodes
  - Windows: call workflows-kotlin\build\nodes\runnodes.bat

Each Spring Boot server needs to be started in its own terminal/command prompt, replace X with A, B and C:

- Unix/Mac OSX: `./gradlew runPartyXServer`
  - Windows: `gradlew.bat runPartyXServer`

Look for the Started ServerKt in X seconds message, don't rely on the % indicator.

**Warning:** On Unix/Mac OSX, do not click/change focus until all seven additional terminal windows have opened, or some nodes may fail to start.

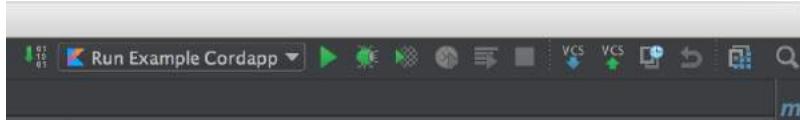
For each node, the `runnodes` script creates a node tab/window:

```
/ _ _ / _ _ _ / _ _ _  
/ / _ _ / _ / _ / _ / _ _ / _ _ _  
\_ _ / _ / _ \_, _ / \_, _ /  
  
--- Corda Open Source corda-4.1 (4157c25) -----  
→---  
  
Logs can be found in : /Users/joeldudley/Desktop/cordapp-example/  
→workflows-kotlin/build/nodes/PartyA/logs  
Database connection url is : jdbc:h2:tcp://localhost:59472/node  
Incoming connection address : localhost:10005  
Listening on port : 10005  
Loaded CorDapps : corda-finance-corda-4.1, cordapp-example-0.  
→1, corda-core-corda-4.1  
Node for "PartyA" started up and registered in 38.59 sec  
  
Welcome to the Corda interactive shell.  
Useful commands include 'help' to see what is available, and 'bye' to shut down the  
→node.  
  
Fri Mar 02 17:34:02 GMT 2018>>>
```

It usually takes around 60 seconds for the nodes to finish starting up. To ensure that all the nodes are running, you can query the ‘status’ end-point located at [http://localhost:\[port\]/api/status](http://localhost:[port]/api/status) (e.g. <http://localhost:50005/api/status> for PartyA).

## Running the example CorDapp from IntelliJ

- Select the Run Example CorDapp – Kotlin run configuration from the drop-down menu at the top right-hand side of the IDE
- Click the green arrow to start the nodes:



- To stop the nodes, press the red square button at the top right-hand side of the IDE, next to the run configurations

### 8.3.4 Interacting with the example CorDapp

#### Via HTTP

The Spring Boot servers run locally on the following ports:

- PartyA: localhost:50005
- PartyB: localhost:50006
- PartyC: localhost:50007

These ports are defined in `clients/build.gradle`.

Each Spring Boot server exposes the following endpoints:

- `/api/example/me`
- `/api/example/peers`
- `/api/example/ious`
- `/api/example/create-iou` with parameters `iouValue` and `partyName` which is CN name of a node

There is also a web front-end served from the home web page e.g. `localhost:50005`.

**Warning:** The content is only available for demonstration purposes and does not implement anti-XSS, anti-XSRF or other security techniques. Do not use this code in production.

#### Creating an IOU via the endpoint

An IOU can be created by sending a PUT request to the `/api/example/create-iou` endpoint directly, or by using the the web form served from the home directory.

To create an IOU between PartyA and PartyB, run the following command from the command line:

```
curl -X PUT 'http://localhost:50005/api/example/create-iou?iouValue=1&
→partyName=O=PartyB,L>New%20York,C=US'
```

Note that both PartyA's port number (50005) and PartyB are referenced in the PUT request path. This command instructs PartyA to agree an IOU with PartyB. Once the process is complete, both nodes will have a signed, notarised copy of the IOU. PartyC will not.

## Submitting an IOU via the web front-end

To create an IOU between PartyA and PartyB, navigate to the home directory for the node, click the “create IOU” button at the top-left of the page, and enter the IOU details into the web-form. The IOU must have a positive value. For example:

```
Counterparty: Select from list
Value (Int): 5
```

And click submit. Upon clicking submit, the modal dialogue will close, and the nodes will agree the IOU.

## Checking the output

Assuming all went well, you can view the newly-created IOU by accessing the vault of PartyA or PartyB:

*Via the HTTP API:*

- PartyA’s vault: Navigate to <http://localhost:50005/api/example/ious>
- PartyB’s vault: Navigate to <http://localhost:50006/api/example/ious>

*Via home page:*

- PartyA: Navigate to <http://localhost:50005> and hit the “refresh” button
- PartyB: Navigate to <http://localhost:50006> and hit the “refresh” button

The vault and web front-end of PartyC (at `localhost:50007`) will not display any IOUs. This is because PartyC was not involved in this transaction.

## Via the interactive shell (terminal only)

Nodes started via the terminal will display an interactive shell:

```
Welcome to the Corda interactive shell.
Useful commands include 'help' to see what is available, and 'bye' to shut down the node.

Fri Jul 07 16:36:29 BST 2017>>>
```

Type `flow list` in the shell to see a list of the flows that your node can run. In our case, this will return the following list:

```
com.example.flow.ExampleFlow$Initiator
net.corda.core.flows.ContractUpgradeFlow$Authorise
net.corda.core.flows.ContractUpgradeFlow$Deauthorise
net.corda.core.flows.ContractUpgradeFlow$Initiate
net.corda.finance.flows.CashExitFlow
net.corda.finance.flows.CashIssueAndPaymentFlow
net.corda.finance.flows.CashIssueFlow
net.corda.finance.flows.CashPaymentFlow
net.corda.finance.internal.CashConfigDataFlow
```

## Creating an IOU via the interactive shell

We can create a new IOU using the `ExampleFlow$Initiator` flow. For example, from the interactive shell of PartyA, you can agree an IOU of 50 with PartyB by running `flow start ExampleFlow$Initiator iouValue: 50, otherParty: "O=PartyB, L>New York, C=US"`.

This will print out the following progress steps:

- ✓ Generating transaction based on new IOU.
- ✓ Verifying contract constraints.
- ✓ Signing transaction with our private key.
- ✓ Gathering the counterparty's signature.
  - ✓ Collecting signatures from counterparties.
  - ✓ Verifying collected signatures.
- ✓ Obtaining notary signature and recording transaction.
  - ✓ Requesting signature by notary service
    - Requesting signature by Notary service
    - Validating response from Notary service
  - ✓ Broadcasting transaction to participants
- ✓ Done

## Checking the output

We can also issue RPC operations to the node via the interactive shell. Type `run` to see the full list of available operations.

You can see the newly-created IOU by running `run vaultQuery contractStateType: com.example.state.IOUState`.

As before, the interactive shell of PartyC will not display any IOUs.

## Via the h2 web console

You can connect directly to your node's database to see its stored states, transactions and attachments. To do so, please follow the instructions in [Node database](#).

### 8.3.5 Running nodes across machines

The nodes can be configured to communicate as a network even when distributed across several machines:

- Deploy the nodes as usual:
  - Unix/Mac OSX: `./gradlew deployNodes`
  - Windows: `gradlew.bat deployNodes`
- Navigate to the build folder (`workflows-kotlin/build/nodes`)
- For each node, open its `node.conf` file and change `localhost` in its `p2pAddress` to the IP address of the machine where the node will be run (e.g. `p2pAddress="10.18.0.166:10007"`)
- These changes require new node-info files to be distributed amongst the nodes. Use the network bootstrapper tool (see [Network Bootstrapper](#)) to update the files and have them distributed locally:  
`java -jar network-bootstrapper.jar workflows-kotlin/build/nodes`

- Move the node folders to their individual machines (e.g. using a USB key). It is important that none of the nodes - including the notary - end up on more than one machine. Each computer should also have a copy of runnodes and runnodes.bat.

For example, you may end up with the following layout:

- Machine 1: Notary, PartyA, runnodes, runnodes.bat
- Machine 2: PartyB, PartyC, runnodes, runnodes.bat
- After starting each node, the nodes will be able to see one another and agree IOUs among themselves

**Warning:** The bootstrapper must be run **after** the node.conf files have been modified, but **before** the nodes are distributed across machines. Otherwise, the nodes will not be able to communicate.

---

**Note:** If you are using H2 and wish to use the same h2port value for two or more nodes, you must only assign them that value after the nodes have been moved to their individual machines. The initial bootstrapping process requires access to the nodes' databases and if two nodes share the same H2 port, the process will fail.

---

### 8.3.6 Testing your CorDapp

Corda provides several frameworks for writing unit and integration tests for CorDapps.

#### Contract tests

You can run the CorDapp's contract tests by running the Run Contract Tests – Kotlin run configuration.

#### Flow tests

You can run the CorDapp's flow tests by running the Run Flow Tests – Kotlin run configuration.

#### Integration tests

You can run the CorDapp's integration tests by running the Run Integration Tests – Kotlin run configuration.

#### Running tests in IntelliJ

See [Running tests in IntelliJ](#)

### 8.3.7 Debugging your CorDapp

See [Debugging a CorDapp](#).

## 8.4 CorDapp samples

There are two distinct sets of samples provided with Corda, one introducing new developers to how to write CorDapps, and more complex worked examples of how solutions to a number of common designs could be implemented in a CorDapp. The former can be found on [the Corda website](#). In particular, new developers should start with the *example CorDapp*.

The advanced samples are contained within the *samples/* folder of the Corda repository. The most generally useful of these samples are:

1. The *trader-demo*, which shows a delivery-vs-payment atomic swap of commercial paper for cash
2. The *attachment-demo*, which demonstrates uploading attachments to nodes
3. The *bank-of-corda-demo*, which shows a node acting as an issuer of assets (the Bank of Corda) while remote client applications request issuance of some cash on behalf of a node called Big Corporation

Documentation on running the samples can be found inside the sample directories themselves, in the *README.md* file.

---

**Note:** If you would like to see flow activity on the nodes type in the node terminal `flow watch`.

---

Please report any bugs with the samples on [GitHub](#).

## 8.5 Structuring a CorDapp

### Contents

- *Structuring a CorDapp*
  - *Modules*
  - *Template CorDapps*
    - \* *Build system*
    - \* *Modules*
      - *Module one - cordapp-contracts-states*
      - *Module two - cordapp*

### 8.5.1 Modules

The source code for a CorDapp is divided into one or more modules, each of which will be compiled into a separate JAR. Together, these JARs represent a single CorDapp. Typically, a Cordapp contains all the classes required for it to be used standalone. However, some Cordapps are only libraries for other Cordapps and cannot be run standalone.

A common pattern is to have:

- One module containing only the CorDapp's contracts and/or states, as well as any required dependencies
- A second module containing the remaining classes that depend on these contracts and/or states

This is because each time a contract is used in a transaction, the entire JAR containing the contract's definition is attached to the transaction. This is to ensure that the exact same contract and state definitions are used when verifying this transaction at a later date. Because of this, you will want to keep this module, and therefore the resulting JAR file, as small as possible to reduce the size of your transactions and keep your node performant.

However, this two-module structure is not prescriptive:

- A library CorDapp containing only contracts and states would only need a single module
- In a CorDapp with multiple sets of contracts and states that **do not** depend on each other, each independent set of contracts and states would go in a separate module to reduce transaction size
- In a CorDapp with multiple sets of contracts and states that **do** depend on each other, either keep them in the same module or create separate modules that depend on each other
- The module containing the flows and other classes can be structured in any way because it is not attached to transactions

### 8.5.2 Template CorDapps

You should base your project on one of the following templates:

- [Java Template CorDapp](#) (for CorDapps written in Java)
- [Kotlin Template CorDapp](#) (for CorDapps written in Kotlin)

Please use the branch of the template that corresponds to the major version of Corda you are using. For example, someone building a CorDapp on Corda 4.1 should use the `release-V4` branch of the template.

#### Build system

The templates are built using Gradle. A Gradle wrapper is provided in the `wrapper` folder, and the dependencies are defined in the `build.gradle` files. See [Building and installing a CorDapp](#) for more information.

No templates are currently provided for Maven or other build systems.

#### Modules

The templates are split into two modules:

- A `cordapp-contracts-states` module containing the contracts and states
- A `cordapp` module containing the remaining classes that depends on the `cordapp-contracts-states` module

These modules will be compiled into two JARs - a `cordapp-contracts-states` JAR and a `cordapp` JAR - which together represent the Template CorDapp.

#### Module one - `cordapp-contracts-states`

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



(continued from previous page)

```

└── 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 of the Java template:

```

.
├── main
│   └── java
│       └── com
│           └── template
│               ├── TemplateApi.java
│               ├── TemplateClient.java
│               ├── TemplateFlow.java
│               ├── TemplateSerializationWhitelist.java
│               └── TemplateWebPlugin.java
│
│   └── resources
│       ├── META-INF
│       │   └── services
│       │       ├── net.corda.core.serialization.SerializationWhitelist
│       │       └── net.corda.webserver.services.WebServerPluginRegistry
│
│       └── certificates
│           └── templateWeb
|
└── test
    └── java
        └── com
            └── template
                ├── ContractTests.java
                ├── FlowTests.java
                └── NodeDriver.java
|
└── integrationTest
    └── java
        └── com
            └── template
                └── DriverBasedTest.java

```

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:

- `java`, which contains the source-code for our CorDapp:

- `TemplateFlow.java`, which contains a template `FlowLogic` subclass
- `TemplateState.java`, which contains a template `ContractState` implementation
- `TemplateContract.java`, which contains a template `Contract` implementation
- `TemplateSerializationWhitelist.java`, which contains a template `SerializationWhitelist` implementation
- `TemplateApi.java`, which contains a template API for the deprecated Corda webserver
- `TemplateWebPlugin.java`, which registers the API and front-end for the deprecated Corda webserver
- `TemplateClient.java`, which contains a template RPC client for interacting with our CorDapp
- `resources/META-INF/services`, which contains various registries:
  - `net.corda.core.serialization.SerializationWhitelist`, which registers the CorDapp's serialisation whitelists
  - `net.corda.webserver.services.WebServerPluginRegistry`, which registers the CorDapp's web plugins
- `resources/templateWeb`, which contains a template front-end

In a production CorDapp:

- We would remove the files related to the deprecated Corda webserver (`TemplateApi.java`, `TemplateWebPlugin.java`, `resources/templateWeb`, and `net.corda.webserver.services.WebServerPluginRegistry`) and replace them with a production-ready webserver
- We would also move `TemplateClient.java` into a separate module so that it is not included in the CorDapp

## 8.6 Building and installing a CorDapp

### Contents

- *Building and installing a CorDapp*
  - *CorDapp format*
  - *Build tools*
  - *Setting your dependencies*
    - \* *Choosing your Corda, Quasar and Kotlin versions*
    - \* *Corda dependencies*
    - \* *Dependencies on other CorDapps*
    - \* *Other dependencies*
    - \* *Signing the CorDapp JAR*
    - \* *Example*
  - *Creating the CorDapp JAR*
  - *Installing the CorDapp JAR*

- *CorDapp configuration files*
  - \* *Using CorDapp configuration with the deployNodes task*
- *Minimum and target platform version*
- *Separation of CorDapp contracts, flows and services*
- *CorDapp Contract Attachments*

CorDapps run on the Corda platform and integrate with it and each other. This article explains how to build CorDapps. To learn what a CorDapp is, please read [What is a CorDapp?](#).

### 8.6.1 CorDapp format

A CorDapp is a semi-fat JAR that contains all of the CorDapp's dependencies *except* the Corda core libraries and any other CorDapps it depends on.

For example, if a Cordapp depends on `corda-core`, `your-other-cordapp` and `apache-commons`, then the Cordapp JAR will contain:

- All classes and resources from the `apache-commons` JAR and its dependencies
- *Nothing* from the other two JARs

### 8.6.2 Build tools

In the instructions that follow, we assume you are using Gradle and the `cordapp` plugin to build your CorDapp. You can find examples of building a CorDapp using these tools in the [Kotlin CorDapp Template](#) and the [Java CorDapp Template](#).

To ensure you are using the correct version of Gradle, you should use the provided Gradle Wrapper by copying across the following folder and files from the [Kotlin CorDapp Template](#) or the [Java CorDapp Template](#) to the root of your project:

- `gradle/`
- `gradlew`
- `gradlew.bat`

### 8.6.3 Setting your dependencies

#### Choosing your Corda, Quasar and Kotlin versions

Several `ext` variables are used in a CorDapp's `build.gradle` file to define version numbers that should match the version of Corda you're developing against:

- `ext.corda_release_version` defines the version of Corda itself
- `ext.corda_gradle_plugins_version` defines the version of the Corda Gradle Plugins
- `ext.quasar_version` defines the version of Quasar, a library that we use to implement the flow framework
- `ext.kotlin_version` defines the version of Kotlin (if using Kotlin to write your CorDapp)

The current versions used are as follows:

```
ext.corda_release_version = '4.1'
ext.corda_gradle_plugins_version = '4.0.42'
ext.quasar_version = '0.7.10'
ext.kotlin_version = '1.2.71'
```

In certain cases, you may also wish to build against the unstable Master branch. See [building-against-master](#).

## Corda dependencies

The `cordapp` plugin adds three new gradle configurations:

- `cordaCompile`, which extends `compile`
- `cordaRuntime`, which extends `runtime`
- `cordapp`, which extends `compile`

`cordaCompile` and `cordaRuntime` indicate dependencies that should not be included in the CorDapp JAR. These configurations should be used for any Corda dependency (e.g. `corda-core`, `corda-node`) in order to prevent a dependency from being included twice (once in the CorDapp JAR and once in the Corda JARs). The `cordapp` dependency is for declaring a compile-time dependency on a “semi-fat” CorDapp JAR in the same way as `cordaCompile`, except that `Cordformation` will only deploy CorDapps contained within the `cordapp` configuration.

Here are some guidelines for Corda dependencies:

- When building a CorDapp, you should always include `net.corda:corda-core:$corda_release_version` as a `cordaCompile` dependency, and `net.corda:corda:$corda_release_version` as a `cordaRuntime` dependency
- When building an RPC client that communicates with a node (e.g. a webserver), you should include `net.corda:corda-rpc:$corda_release_version` as a `cordaCompile` dependency.
- When you need to use the network bootstrapper to bootstrap a local network (e.g. when using `Cordformation`), you should include `net.corda:corda-node-api:$corda_release_version` as either a `cordaRuntime` or a `runtimeOnly` dependency. You may also wish to include an implementation of SLF4J as a `runtimeOnly` dependency for the network bootstrapper to use.
- To use Corda’s test frameworks, add `net.corda:corda-test-utils:$corda_release_version` as a `testCompile` dependency. Never include `corda-test-utils` as a `compile` or `cordaCompile` dependency.
- Any other Corda dependencies you need should be included as `cordaCompile` dependencies.

Here is an overview of the various Corda dependencies:

- `corda` - The Corda fat JAR. Do not use as a `compile` dependency. Required as a `cordaRuntime` dependency when using `Cordformation`
- `corda-confidential-identities` - A part of the core Corda libraries. Automatically pulled in by other libraries
- `corda-core` - Usually automatically included by another dependency, contains core Corda utilities, model, and functionality. Include manually if the utilities are useful or you are writing a library for Corda
- `corda-core-deterministic` - Used by the Corda node for deterministic contracts. Not likely to be used externally
- `corda-djvm` - Used by the Corda node for deterministic contracts. Not likely to be used externally

- `corda-finance-contracts`, `corda-finance-workflows` and deprecated `corda-finance`. Corda finance CorDapp, use contracts and flows parts respectively. Only include as a `cordaCompile` dependency if using as a dependent Cordapp or if you need access to the Corda finance types. Use as a `cordapp` dependency if using as a CorDapp dependency (see below)
- `corda-jackson` - Corda Jackson support. Use if you plan to serialise Corda objects to and/or from JSON
- `corda-jfx` - JavaFX utilities with some Corda-specific models and utilities. Only use with JavaFX apps
- `corda-mock` - A small library of useful mocks. Use if the classes are useful to you
- `corda-node` - The Corda node. Do not depend on. Used only by the Corda fat JAR and indirectly in testing frameworks. (If your CorDapp `_must_` depend on this for some reason then it should use the `compileOnly` configuration here - but please don't do this if you can possibly avoid it!)
- `corda-node-api` - The node API. Required to bootstrap a local network
- `corda-node-driver` - Testing utility for programmatically starting nodes from JVM languages. Use for tests
- `corda-rpc` - The Corda RPC client library. Used when writing an RPC client
- `corda-serialization` - The Corda core serialization library. Automatically included by other dependencies
- `corda-serialization-deterministic` - The Corda core serialization library. Automatically included by other dependencies
- `corda-shell` - Used by the Corda node. Never depend on directly
- `corda-test-common` - A common test library. Automatically included by other test libraries
- `corda-test-utils` - Used when writing tests against Corda/Cordapps
- `corda-tools-explorer` - The Node Explorer tool. Do not depend on
- `corda-tools-network-bootstrapper` - The Network Builder tool. Useful in build scripts
- `corda-tools-shell-cli` - The Shell CLI tool. Useful in build scripts
- `corda-webserver-impl` - The Corda webserver fat JAR. Deprecated. Usually only used by build scripts
- `corda-websever` - The Corda webserver library. Deprecated. Use a standard webserver library such as Spring instead

### Dependencies on other CorDapps

Your CorDapp may also depend on classes defined in another CorDapp, such as states, contracts and flows. There are two ways to add another CorDapp as a dependency in your CorDapp's `build.gradle` file:

- `cordapp project (" :another-cordapp")` (use this if the other CorDapp is defined in a module in the same project)
- `cordapp "net.corda:another-cordapp:1.0"` (use this otherwise)

The `cordapp` gradle configuration serves two purposes:

- When using the `cordformation` Gradle plugin, the `cordapp` configuration indicates that this JAR should be included on your node as a CorDapp
- When using the `cordapp` Gradle plugin, the `cordapp` configuration prevents the dependency from being included in the CorDapp JAR

Note that the `cordformation` and `cordapp` Gradle plugins can be used together.

## Other dependencies

If your CorDapps have any additional external dependencies, they can be specified like normal Kotlin/Java dependencies in Gradle. See the example below, specifically the `apache-commons` include.

For further information about managing dependencies, see the [Gradle docs](#).

## **S**igning the CorDapp JAR

The `cordapp` plugin can sign the generated CorDapp JAR file using [JAR signing and verification tool](#). Signing the CorDapp enables its contract classes to use signature constraints instead of other types of the constraints, for constraints explanation refer to [API: Contract Constraints](#). By default the JAR file is signed by Corda development certificate. The signing process can be disabled or configured to use an external keystore. The `signing` entry may contain the following parameters:

- `enabled` the control flag to enable signing process, by default is set to `true`, set to `false` to disable signing
- `options` any relevant parameters of [SignJar ANT task](#), by default the JAR file is signed with Corda development key, the external keystore can be specified, the minimal list of required options is shown below, for other options referer to [SignJar task](#):
  - `keystore` the path to the keystore file, by default `cordadevcakeys.jks` keystore is shipped with the plugin
  - `alias` the alias to sign under, the default value is `cordaintermediateca`
  - `storepass` the keystore password, the default value is `cordacadevpass`
  - `keypass` the private key password if it's different than the password for the keystore, the default value is `cordacadevkeypass`
  - `storetype` the keystore type, the default value is `JKS`

The parameters can be also set by system properties passed to Gradle build process. The system properties should be named as the relevant option name prefixed with '`signing.`', e.g. a value for `alias` can be taken from the `signing.alias` system property. The following system properties can be used: `signing.enabled`, `signing.keystore`, `signing.alias`, `signing.storepass`, `signing.keypass`, `signing.storetype`. The resolution order of a configuration value is as follows: the signing process takes a value specified in the `signing` entry first, the empty string "" is also considered as the correct value. If the option is not set, the relevant system property named `signing.option` is tried. If the system property is not set then the value defaults to the configuration of the Corda development certificate.

The example `cordapp` plugin with plugin signing configuration:

```
cordapp {
    signing {
        enabled true
        options {
            keystore "/path/to/jarSignKeystore.p12"
            alias "cordapp-signer"
            storepass "secret1!"
            keypass "secret1!"
            storetype "PKCS12"
        }
    }
} //...
```

CorDapp auto-signing allows to use signature constraints for contracts from the CorDapp without need to create a keystore and configure the `cordapp` plugin. For production deployment ensure to sign the CorDapp using your own certificate e.g. by setting system properties to point to an external keystore or by disabling signing in `cordapp`

plugin and signing the CordDapp JAR downstream in your build pipeline. CorDapp signed by Corda development certificate is accepted by Corda node only when running in the development mode. In case CordDapp signed by the (default) development key is run on node in the production mode (e.g. for testing), the node may be set to accept the development key by adding the `cordappSignerKeyFingerprintBlacklist = []` property set to empty list (see [Configuring a node](#)).

Signing options can be contextually overwritten by the relevant system properties as described above. This allows the single `build.gradle` file to be used for a development build (defaulting to the Corda development keystore) and for a production build (using an external keystore). The example system properties setup for the build process which overrides signing options:

```
./gradlew -Dsigning.keystore="/path/to/keystore.jks" -Dsigning.alias="alias" -  
-Dsigning.storepass="password" -Dsigning.keypass="password"
```

Without providing the system properties, the build will sign the CorDapp with the default Corda development keystore:

```
./gradlew
```

CorDapp signing can be disabled for a build:

```
./gradlew -Dsigning.enabled=false
```

Other system properties can be explicitly assigned to options by calling `System.getProperty` in `cordapp` plugin configuration. For example the below configuration sets the specific signing algorithm when a system property is available otherwise defaults to an empty string:

```
cordapp {  
    signing {  
        options {  
            sigalg System.getProperty('custom.sigalg', '')  
        }  
    }  
    //...  
}
```

Then the build process can set the value for `custom.sigalg` system property and other system properties recognized by `cordapp` plugin:

```
./gradlew -Dcustom.sigalg="SHA256withECDSA" -Dsigning.keystore="/path/to/keystore.jks"  
-Dsigning.alias="alias" -Dsigning.storepass="password" -Dsigning.keypass="password"
```

To check if CorDapp is signed use JAR signing and verification tool:

```
jarsigner --verify path/to/cordapp.jar
```

Cordformation plugin can also sign CorDapps JARs, when deploying set of nodes, see [Creating nodes locally](#).

If your build system post-processes the Cordapp JAR, then the modified JAR content may be out-of-date or not complete with regards to a signature file. In this case you can sign the Cordapp as a separate step and disable the automatic signing by the `cordapp` plugin. The `cordapp` plugin contains a standalone task `signJar` which uses the same signing configuration. The task has two parameters: `inputJars` - to pass JAR files to be signed and an optional `postfix` which is added to the name of signed JARs (it defaults to “-signed”). The signed JARs are returned as `outputJars` property.

For example in order to sign a JAR modified by `modifyCordapp` task, create an instance of the `net.corda.plugins.SignJar` task (below named as `sign`). The output of `modifyCordapp` task is passed to `inputJars` and the `sign` task is run after `modifyCordapp` one:

```

task sign(type: net.corda.plugins.SignJar) {
    inputJars modifyCordapp
}
modifyCordapp.finalizedBy sign
cordapp {
    signing {
        enabled false
    }
    /**
 */
}

```

The task creates a new JAR file named `*-signed.jar` which should be used further in your build/publishing process. Also the best practice is to disable signing by the `cordapp` plugin as shown in the example.

## Example

Below is a sample CorDapp Gradle dependencies block. When building your own CorDapp, use the `build.gradle` file of the [Kotlin CorDapp Template](#) or the [Java CorDapp Template](#) as a starting point.

```

dependencies {
    // Corda integration dependencies
    cordaCompile "net.corda:corda-core:$corda_release_version"
    cordaCompile "net.corda:corda-finance-contracts:$corda_release_version"
    cordaCompile "net.corda:corda-finance-workflows:$corda_release_version"
    cordaCompile "net.corda:corda-jackson:$corda_release_version"
    cordaCompile "net.corda:corda-rpc:$corda_release_version"
    cordaCompile "net.corda:corda-node-api:$corda_release_version"
    cordaCompile "net.corda:corda-webserver-impl:$corda_release_version"
    cordaRuntime "net.corda:corda:$corda_release_version"
    cordaRuntime "net.corda:corda-webserver:$corda_release_version"
    testCompile "net.corda:corda-test-utils:$corda_release_version"

    // Corda Plugins: dependent flows and services
    // Identifying a CorDapp by its module in the same project.
    cordapp project(":cordapp-contracts-states")
    // Identifying a CorDapp by its fully-qualified name.
    cordapp "net.corda:bank-of-corda-demo:1.0"

    // Some other dependencies
    compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    testCompile "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
    testCompile "junit:junit:$junit_version"

    compile "org.apache.commons:commons-lang3:3.6"
}

```

### 8.6.4 Creating the CorDapp JAR

Once your dependencies are set correctly, you can build your CorDapp JAR(s) using the Gradle `jar` task

- Unix/Mac OSX: `./gradlew jar`
- Windows: `gradlew.bat jar`

Each of the project's modules will be compiled into its own CorDapp JAR. You can find these CorDapp JARs in the `build/libs` folders of each of the project's modules.

**Warning:** The hash of the generated CorDapp JAR is not deterministic, as it depends on variables such as the timestamp at creation. Nodes running the same CorDapp must therefore ensure they are using the exact same CorDapp JAR, and not different versions of the JAR created from identical sources.

The filename of the JAR must include a unique identifier to deduplicate it from other releases of the same CorDapp. This is typically done by appending the version string to the CorDapp's name. This unique identifier should not change once the JAR has been deployed on a node. If it does, make sure no one is relying on `FlowContext.appName` in their flows (see [Versioning](#)).

## 8.6.5 Installing the CorDapp JAR

---

**Note:** Before installing a CorDapp, you must create one or more nodes to install it on. For instructions, please see [Creating nodes locally](#).

---

At start-up, nodes will load any CorDapps present in their `cordapps` folder. In order to install a CorDapp on a node, the CorDapp JAR must be added to the `<node_dir>/cordapps/` folder (where `node_dir` is the folder in which the node's JAR and configuration files are stored) and the node restarted.

## 8.6.6 CorDapp configuration files

CorDapp configuration files should be placed in `<node_dir>/cordapps/config`. The name of the file should match the name of the JAR of the CorDapp (eg; if your CorDapp is called `hello-0.1.jar` the config should be `config/hello-0.1.conf`).

Config files are currently only available in the [Typesafe/Lightbend](#) config format. These files are loaded during node startup.

CorDapp configuration can be accessed from `CordappContext::config` whenever a `CordappContext` is available. For example:

```
@StartableByRPC
class GetStringConfigFlow(private val configKey: String) : FlowLogic<String>() {
    object READING : ProgressTracker.Step("Reading config")
    override val progressTracker = ProgressTracker(READING)

    @Suspendable
    override fun call(): String {
        progressTracker.currentStep = READING
        val config = serviceHub.getApplicationContext().config
        return config.getString(configKey)
    }
}
```

### Using CorDapp configuration with the `deployNodes` task

If you want to generate CorDapp configuration when using the `deployNodes` Gradle task, then you can use the `cordapp` or `projectCordapp` properties on the node. For example:

```

task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    nodeDefaults {
        // this external CorDapp will be included in each project
        cordapp("$corda_release_group:corda-finance-contracts:$corda_release_version")
        // this external CorDapp will be included in each project with the given config
        cordapp("$corda_release_group:corda-finance-workflows:$corda_release_version")
    }
    node {
        name "O=Bank A,L=London,C=GB"
        ...
        // This adds configuration for another CorDapp project within the build
        cordapp(project(':my-project:workflow-cordapp')) {
            config "someStringValue=test"
        }
        cordapp(project(':my-project:another-cordapp')) {
            // Use a multiline string for complex configuration
            config """
                someStringValue=test
                anotherStringValue=10
            """
        }
    }
}

```

There is an example project that demonstrates this in the `samples` folder of the Corda Git repository, called `cordapp-configuration`. API documentation can be found at [api/kotlin/corda/net.corda.core.cordapp/index.html](#).

### 8.6.7 Minimum and target platform version

CorDapps can advertise their minimum and target platform version. The minimum platform version indicates that a node has to run at least this version in order to be able to run this CorDapp. The target platform version indicates that a CorDapp was tested with this version of the Corda Platform and should be run at this API level if possible. It provides a means of maintaining behavioural compatibility for the cases where the platform's behaviour has changed. These attributes are specified in the JAR manifest of the CorDapp, for example:

```
'Min-Platform-Version': 4
'Target-Platform-Version': 4
```

#### Defaults

- Target-Platform-Version (mandatory) is a whole number and must comply with the rules mentioned above.
- Min-Platform-Version (optional) will default to 1 if not specified.

Using the *cordapp* Gradle plugin, this can be achieved by putting this in your CorDapp's *build.gradle*:

```
cordapp {  
    targetPlatformVersion 4  
    minimumPlatformVersion 4  
}
```

## 8.6.8 Separation of CorDapp contracts, flows and services

It is recommended that **contract** code (states, commands, verification logic) be packaged separately from **business flows** (and associated services). This decoupling enables *contracts* to evolve independently from the *flows* and *services* that use them. Contracts may even be specified and implemented by different providers (eg. Corda currently ships with a cash financial contract which in turn is used in many other flows and many other CorDapps).

As of Corda 4, CorDapps can explicitly differentiate their type by specifying the following attributes in the JAR manifest:

```
'Cordapp-Contract-Name'  
'Cordapp-Contract-Version'  
'Cordapp-Contract-Vendor'  
'Cordapp-Contract-Licence'  
  
'Cordapp-Workflow-Name'  
'Cordapp-Workflow-Version'  
'Cordapp-Workflow-Vendor'  
'Cordapp-Workflow-Licence'
```

### Defaults

*Cordapp-Contract-Name* (optional) if specified, the following Contract related attributes are also used:

- *Cordapp-Contract-Version* (mandatory), must be a whole number starting from 1.
- *Cordapp-Contract-Vendor* (optional), defaults to UNKNOWN if not specified.
- *Cordapp-Contract-Licence* (optional), defaults to UNKNOWN if not specified.

*Cordapp-Workflow-Name* (optional) if specified, the following Workflow related attributes are also used:

- *Cordapp-Workflow-Version* (mandatory), must be a whole number starting from 1.
- *Cordapp-Workflow-Vendor* (optional), defaults to UNKNOWN if not specified.
- *Cordapp-Workflow-Licence* (optional), defaults to UNKNOWN if not specified.

As with the general CorDapp attributes (minimum and target platform version), these can be specified using the Gradle *cordapp* plugin as follows:

For a contract only CorDapp we specify the *contract* tag:

```
cordapp {  
    targetPlatformVersion 4  
    minimumPlatformVersion 3  
    contract {  
        name "my contract name"  
        versionId 1  
        vendor "my company"  
        licence "my licence"  
    }  
}
```

For a CorDapp that contains flows and/or services we specify the *workflow* tag:

```
cordapp {
    targetPlatformVersion 4
    minimumPlatformVersion 3
    workflow {
        name "my workflow name"
        versionId 1
        vendor "my company"
        licence "my licence"
    }
}
```

---

**Note:** It is possible, but *not recommended*, to include everything in a single CorDapp jar and use both the `contract` and `workflow` Gradle plugin tags.

---

**Warning:** Contract states may optionally specify a custom schema mapping (by implementing the `Queryable` interface) in its *contracts* JAR. However, any associated database schema definition scripts (eg. Liquibase change set XML files) must currently be packaged in the *flows* JAR. This is because the node requires access to these schema definitions upon start-up (*contract* JARs are now loaded in a separate attachments classloader). This split also caters for scenarios where the same *contract* CorDapp may wish to target different database providers (and thus, the associated schema DDL may vary to use native features of a particular database). The finance CorDapp provides an illustration of this packaging convention. Future versions of Corda will de-couple this custom schema dependency to remove this anomaly.

## 8.6.9 CorDapp Contract Attachments

As of Corda 4, CorDapp Contract JARs must be installed on a node by a trusted uploader, either by

- installing manually as per [Installing the CorDapp JAR](#) and re-starting the node.
- uploading the attachment JAR to the node via RPC, either programmatically (see [Connecting to a node via RPC](#)) or via the [Node shell](#) by issuing the following command:

```
>>> run uploadAttachment jar: path/to/the/file.jar
```

Contract attachments that are received from a peer over the p2p network are considered **untrusted** and will throw a *UntrustedAttachmentsException* exception when processed by a listening flow that cannot resolve that attachment from its local attachment storage. The flow will be aborted and sent to the nodes flow hospital for recovery and retry. The untrusted attachment JAR will be stored in the nodes local attachment store for review by a node operator. It can be downloaded for viewing using the following CRaSH shell command:

```
>>> run openAttachment id: <hash of untrusted attachment given by  
'UntrustedAttachmentsException' exception
```

Should the node operator deem the attachment trustworthy, they may then issue the following CRaSH shell command to reload it as trusted:

```
>>> run uploadAttachment jar: path/to/the/trusted-file.jar
```

and subsequently retry the failed flow (currently this requires a node re-start).

---

**Note:** this behaviour is to protect the node from executing contract code that was not vetted. It is a temporary precaution until the Deterministic JVM is integrated into Corda whereby execution takes place in a sandboxed environment

which protects the node from malicious code.

---

## 8.7 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.

### 8.7.1 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

### 8.7.2 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 `isDebugEnabled` 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

### 8.7.3 By enabling remote debugging on a node

See [Enabling remote debugging](#).

## 8.8 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 features are available to them. Each Corda release uses the standard semantic versioning scheme of `major.minor`. 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.

### 8.8.1 Platform version

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 suggest migration away from old APIs. In very rare situations APIs may have to be changed, for example due to security issues. There is no relationship between the platform version and the release version - a change in the major or minor values may or may not increase the platform version. However we do endeavour to keep them synchronised for now, as a convenience.

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 it will check its own version against the minimum version requirement for the network.

## 8.8.2 Minimum platform version

Applications can advertise a *minimum platform version* they require. If your app uses new APIs that were added in (for example) Corda 5, you should specify a minimum version of 5. This will ensure the app won't be loaded by older nodes. If you can *optionally* use the new APIs, you can keep the minimum set to a lower number. Attempting to use new APIs on older nodes can cause `NoSuchMethodError` exceptions and similar problems, so you'd want to check the node version using `ServiceHub.myInfo`.

## 8.8.3 Target version

Applications can also advertise a *target version*. This is similar to the concept of the same name in Android and iOS. Apps should advertise the highest version of the platform they have been tested against. This allows the node to activate or deactivate backwards compatibility codepaths depending on whether they're necessary or not, as workarounds for apps designed for earlier versions.

For example, consider an app that uses new features introduced in Corda 4, but which has passed regression testing on Corda 5. It will advertise a minimum platform version of 4 and a target version of 5. These numbers are published in the JAR manifest file.

If this app is loaded into a Corda 6 node, that node may implement backwards compatibility workarounds for your app that make it slower, less secure, or less featureful. You can opt-in to getting the full benefits of the upgrade by changing your target version to 6. By doing this, you promise that you understood all the changes in Corda 6 and have thoroughly tested your app to prove it works. This testing should include ensuring that the app exhibits the correct behaviour on a node running at the new target version, and that the app functions correctly in a network of nodes running at the same target version.

Target versioning is one of the mechanisms we have to keep the platform evolving and improving, without being permanently constrained to being bug-for-bug compatible with old versions. When no apps are loaded that target old versions, any emulations of older bugs or problems can be disabled.

## 8.8.4 Publishing versions in your JAR manifests

A well structured CorDapp should be split into two separate modules:

1. A contracts jar, that contains your states and contract logic.
2. A workflows jar, that contains your flows, services and other support libraries.

The reason for this split is that the contract JAR will be attached to transactions and sent around the network, because this code is what defines the data structures and smart contract logic all nodes will validate. If the rest of your app is a part of that same JAR, it'll get sent around the network too even though it's not needed and will never be used. By splitting your app into a contracts JAR and a workflows JAR that depends on the contracts JAR, this problem is avoided.

In the `build.gradle` file for your contract module, add a block like this:

```
cordapp {  
    targetPlatformVersion 5  
    minimumPlatformVersion 4  
    contract {  
        name "MegaApp Contracts"  
        vendor "MegaCorp"  
        licence "MegaLicence"  
        versionId 1  
    }  
}
```

This will put the necessary entries into your JAR manifest to set both platform version numbers. If they aren't specified, both default to 1. Your app can itself have a version number, which should always increment and must also always be an integer.

And in the `build.gradle` file for your workflows jar, add a block like this:

```
cordapp {
    targetPlatformVersion 5
    minimumPlatformVersion 4
    workflow {
        name "MegaApp"
        vendor "MegaCorp"
        licence "MegaLicence"
        versionId 1
    }
}
```

It's entirely expected and reasonable to have an open source contracts module and a proprietary workflow module - the latter may contain sophisticated or proprietary business logic, machine learning models, even user interface code. There's nothing that restricts it to just being Corda flows or services.

---

**Important:** The `versionId` specified for the JAR manifest is checked by the platform and is used for informative purposes only. See "["App versioning with Signature Constraints"](#)" for more information.

---

**Note:** You can read the original design doc here: [design/targetversion/design](#).

---

## 8.9 Release new CorDapp versions

---

**Note:** This document only concerns the upgrading of CorDapps and not the Corda platform itself (wire format, node database schemas, etc.).

---

### Contents

- *Release new CorDapp versions*
  - *CorDapp versioning*
  - *Flow versioning*
    - \* *Defining a flow's interface*
    - \* *Non-backwards compatible flow changes*
    - \* *Consequences of running flows with incompatible versions*
    - \* *Ensuring flow backwards-compatibility*
    - \* *Handling interface changes to inlined subflows*
    - \* *Performing flow upgrades*
    - \* *Draining the node*

- *Contract and state versioning*
  - \* *Performing explicit contract and state upgrades*
    - 1. *Preserve the existing state and contract definitions*
    - 2. *Write the new state and contract definitions*
    - 3. *Create the new CorDapp JAR*
    - 4. *Distribute the new CorDapp JAR*
    - 5. *Stop the nodes*
    - 6. *Re-run the network bootstrapper (only if you want to whitelist the new contract)*
    - 7. *Restart the nodes*
    - 8. *Authorise the upgrade*
    - 9. *Perform the upgrade*
    - 10. *Migrate the new upgraded state to the Signature Constraint from the zone constraint*
  - \* *Points to note*
    - *Capabilities of the contract upgrade flows*
    - *Logistics*
- *State schema versioning*
- *Serialisation*
  - \* *The Corda serialisation format*
  - \* *Writing classes that meet the serialisation format requirements*
- *Testing CorDapp upgrades*

### 8.9.1 CorDapp versioning

The Corda platform does not mandate a version number on a per-CorDapp basis. Different elements of a CorDapp are allowed to evolve separately. Sometimes, however, a change to one element will require changes to other elements. For example, changing a shared data structure may require flow changes that are not backwards-compatible.

### 8.9.2 Flow versioning

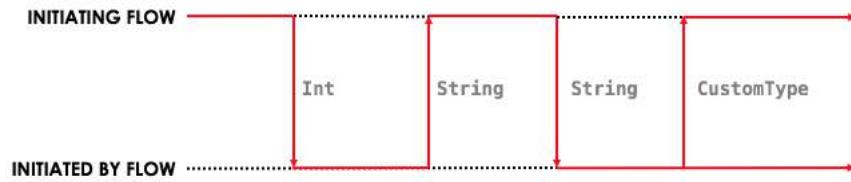
Any flow that initiates other flows must be annotated with the `@InitiatingFlow` annotation, which is defined as:

```
annotation class InitiatingFlow(val version: Int = 1)
```

The `version` property, which defaults to 1, specifies the flow's version. This integer value should be incremented whenever there is a release of a flow which has changes that are not backwards-compatible. A non-backwards compatible change is one that changes the interface of the flow.

#### Defining a flow's interface

The flow interface is defined by the sequence of `send` and `receive` calls between an `InitiatingFlow` and an `InitiatedBy` flow, including the types of the data sent and received. We can picture a flow's interface as follows:



In the diagram above, the `InitiatingFlow`:

- Sends an `Int`
- Receives a `String`
- Sends a `String`
- Receives a `CustomType`

The `InitiatedBy` flow does the opposite:

- Receives an `Int`
- Sends a `String`
- Receives a `String`
- Sends a `CustomType`

As long as both the `InitiatingFlow` and the `InitiatedBy` flows conform to the sequence of actions, the flows can be implemented in any way you see fit (including adding proprietary business logic that is not shared with other parties).

### Non-backwards compatible flow changes

A flow can become backwards-incompatible in two main ways:

- The sequence of `send` and `receive` calls changes:
  - A `send` or `receive` is added or removed from either the `InitiatingFlow` or `InitiatedBy` flow
  - The sequence of `send` and `receive` calls changes
- The types of the `send` and `receive` calls changes

### Consequences of running flows with incompatible versions

Pairs of `InitiatingFlow` flows and `InitiatedBy` flows that have incompatible interfaces are likely to exhibit the following behaviour:

- The flows hang indefinitely and never terminate, usually because a flow expects a response which is never sent from the other side
- One of the flow ends with an exception: “Expected Type X but Received Type Y”, because the `send` or `receive` types are incorrect
- One of the flows ends with an exception: “Counterparty flow terminated early on the other side”, because one flow sends some data to another flow, but the latter flow has already ended

## Ensuring flow backwards-compatibility

The `InitiatingFlow` version number is included in the flow session handshake and exposed to both parties via the `FlowLogic.getFlowContext` method. This method takes a `Party` and returns a `FlowContext` object which describes the flow running on the other side. In particular, it has a `flowVersion` property which can be used to programmatically evolve flows across versions. For example:

```
@Suspendable
override fun call() {
    val otherFlowVersion = otherSession.getCounterpartyFlowInfo().flowVersion
    val receivedString = if (otherFlowVersion == 1) {
        otherSession.receive<Int>().unwrap { it.toString() }
    } else {
        otherSession.receive<String>().unwrap { it }
    }
}
```

```
@Suspendable
@Overide public Void call() throws FlowException {
    int otherFlowVersion = otherSession.getCounterpartyFlowInfo().getFlowVersion();
    String receivedString;

    if (otherFlowVersion == 1) {
        receivedString = otherSession.receive(Integer.class).unwrap(integer -> {
            return integer.toString();
        });
    } else {
        receivedString = otherSession.receive(String.class).unwrap(string -> {
            return string;
        });
    }

    return null;
}
```

This code shows a flow that in its first version expected to receive an `Int`, but in subsequent versions was modified to expect a `String`. This flow is still able to communicate with parties that are running the older CorDapp containing the older flow.

## Handling interface changes to inlined subflows

Here is an example of an in-lined subflow:

```
@StartableByRPC
@InitiatingFlow
class FlowA(val recipient: Party) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        subFlow(FlowB(recipient))
    }
}

@InitiatedBy(FlowA::class)
class FlowC(val otherSession: FlowSession) : FlowLogic() {
    // Omitted.
}
```

(continues on next page)

(continued from previous page)

```
// Note: No annotations. This is used as an inlined subflow.
class FlowB(val recipient: Party) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        val message = "I'm an inlined subflow, so I inherit the @InitiatingFlow's_
        ↪session ID and type."
        initiateFlow(recipient).send(message)
    }
}
```

```
@StartableByRPC
@InitiatingFlow
class FlowA extends FlowLogic<Void> {
    private final Party recipient;

    public FlowA(Party recipient) {
        this.recipient = recipient;
    }

    @Suspendable
    @Override public Void call() throws FlowException {
        subFlow(new FlowB(recipient));

        return null;
    }
}

@InitiatedBy(FlowA.class)
class FlowC extends FlowLogic<Void> {
    // Omitted.
}

// Note: No annotations. This is used as an inlined subflow.
class FlowB extends FlowLogic<Void> {
    private final Party recipient;

    public FlowB(Party recipient) {
        this.recipient = recipient;
    }

    @Suspendable
    @Override public Void call() {
        String message = "I'm an inlined subflow, so I inherit the @InitiatingFlow's_
        ↪session ID and type.";
        initiateFlow(recipient).send(message);

        return null;
    }
}
```

Inlined subflows are treated as being the flow that invoked them when initiating a new flow session with a counterparty. Suppose flow A calls inlined subflow B, which, in turn, initiates a session with a counterparty. The `FlowLogic` type used by the counterparty to determine which counter-flow to invoke is determined by A, and not by B. This means that the response logic for the inlined flow must be implemented explicitly in the `InitiatedBy` flow. This can be done either by calling a matching inlined counter-flow, or by implementing the other side explicitly in the initiated parent

flow. Inlined subflows also inherit the session IDs of their parent flow.

As such, an interface change to an inlined subflow must be considered a change to the parent flow interfaces.

An example of an inlined subflow is `CollectSignaturesFlow`. It has a response flow called `SignTransactionFlow` that isn't annotated with `InitiatedBy`. This is because both of these flows are inlined. How these flows speak to one another is defined by the parent flows that call `CollectSignaturesFlow` and `SignTransactionFlow`.

In code, inlined subflows appear as regular `FlowLogic` instances without either an `InitiatingFlow` or an `InitiatedBy` annotation.

Inlined flows are not versioned, as they inherit the version of their parent `InitiatingFlow` or `InitiatedBy` flow.

Flows which are not an `InitiatingFlow` or `InitiatedBy` flow, or inlined subflows that are not called from an `InitiatingFlow` or `InitiatedBy` flow, can be updated without consideration of backwards-compatibility. Flows of this type include utility flows for querying the vault and flows for reaching out to external systems.

### Performing flow upgrades

1. Update the flow and test the changes. Increment the flow version number in the `InitiatingFlow` annotation
2. Ensure that all versions of the existing flow have finished running and there are no pending `SchedulableFlows` on any of the nodes on the business network. This can be done by [Draining the node](#)
3. Shut down the node
4. Replace the existing CorDapp JAR with the CorDapp JAR containing the new flow
5. Start the node

If you shut down all nodes and upgrade them all at the same time, any incompatible change can be made.

In situations where some nodes may still be using previous versions of a flow and thus new versions of your flow may talk to old versions, the updated flows need to be backwards-compatible. This will be the case for almost any real deployment in which you cannot easily coordinate the roll-out of new code across the network.

### Draining the node

A flow *checkpoint* is a serialised snapshot of the flow's stack frames and any objects reachable from the stack. Checkpoints are saved to the database automatically when a flow suspends or resumes, which typically happens when sending or receiving messages. A flow may be replayed from the last checkpoint if the node restarts. Automatic checkpointing is an unusual feature of Corda and significantly helps developers write reliable code that can survive node restarts and crashes. It also assists with scaling up, as flows that are waiting for a response can be flushed from memory.

However, this means that restoring an old checkpoint to a new version of a flow may cause resume failures. For example if you remove a local variable from a method that previously had one, then the flow engine won't be able to figure out where to put the stored value of the variable.

For this reason, in currently released versions of Corda you must *drain the node* before doing an app upgrade that changes `@Suspendable` code. A drain blocks new flows from starting but allows existing flows to finish. Thus once a drain is complete there should be no outstanding checkpoints or running flows. Upgrading the app will then succeed.

A node can be drained or undrained via RPC using the `setFlowsDrainingModeEnabled` method, and via the shell using the standard `run` command to invoke the RPC. See [Node shell](#) to learn more.

### 8.9.3 Contract and state versioning

There are two types of contract/state upgrade:

1. *Implicit*: By allowing multiple implementations of the contract ahead of time, using constraints. See [API: Contract Constraints](#) to learn more
2. *Explicit*: By creating a special *contract upgrade transaction* and getting all participants of a state to sign it using the contract upgrade flows

The general recommendation for Corda 4 is to use **implicit** upgrades for the reasons described [here](#).

#### Performing explicit contract and state upgrades

In an explicit upgrade, contracts and states can be changed in arbitrary ways, if and only if all of the state's participants agree to the proposed upgrade. To ensure the continuity of the chain the upgraded contract needs to declare the contract and constraint of the states it's allowed to replace.

**Warning:** In Corda 4 we've introduced the Signature Constraint (see [API: Contract Constraints](#)). States created or migrated to the Signature Constraint can't be explicitly upgraded using the Contract upgrade transaction. This feature might be added in a future version. Given the nature of the Signature constraint there should be little need to create a brand new contract to fix issues in the old contract.

#### 1. Preserve the existing state and contract definitions

Currently, all nodes must **permanently** keep **all** old state and contract definitions on their node's classpath if the explicit upgrade process was used on them.

---

**Note:** This requirement will go away in a future version of Corda. In Corda 4, the contract-code-as-attachment feature was implemented only for "normal" transactions. Contract Upgrade and Notary Change transactions will still be executed within the node classpath.

---

#### 2. Write the new state and contract definitions

Update the contract and state definitions. There are no restrictions on how states are updated. However, upgraded contracts must implement the `UpgradedContract` interface. This interface is defined as:

```
interface UpgradedContract<in OldState : ContractState, out NewState : ContractState> {
    <--: Contract {
        val legacyContract: ContractClassName
        fun upgrade(state: OldState): NewState
    }
}
```

The `upgrade` method describes how the old state type is upgraded to the new state type.

By default this new contract will only be able to upgrade legacy states which are constrained by the zone whitelist (see [API: Contract Constraints](#)).

---

**Note:** The requirement for a `legacyContractConstraint` arises from the fact that when a transaction chain is verified and a Contract Upgrade is encountered on the back chain, the verifier wants to know that a legitimate state was transformed into the new contract. The `legacyContractConstraint` is the mechanism by which

this is enforced. Using it, the new contract is able to narrow down what constraint the states it is upgrading should have. If a malicious party would create a fake `com.megacorp.MegaToken` state, he would not be able to use the usual `MegaToken` code as his fake token will not validate because the constraints will not match. The `com.megacorp.SuperMegaToken` would know that it is a fake state and thus refuse to upgrade it. It is safe to omit the `legacyContractConstraint` for the zone whitelist constraint, because the chain of trust is ensured by the Zone operator who would have whitelisted both contracts and checked them.

---

If the hash constraint is used, the new contract should implement `UpgradedContractWithLegacyConstraint` instead, and specify the constraint explicitly:

```
interface UpgradedContractWithLegacyConstraint<in OldState : ContractState, out_
    ↪NewState : ContractState> : UpgradedContract<OldState, NewState> {
    val legacyContractConstraint: AttachmentConstraint
}
```

For example, in case of hash constraints the hash of the legacy JAR file should be provided:

```
override val legacyContractConstraint: AttachmentConstraint
    get() = HashAttachmentConstraint(SecureHash.parse(
        ↪"E02BD2B9B010BBCE49C0D7C35BECEF2C79BEB2EE80D902B54CC9231418A4FA0C" ))
```

### 3. Create the new CorDapp JAR

Produce a new CorDapp JAR file. This JAR file should only contain the new contract and state definitions.

### 4. Distribute the new CorDapp JAR

Place the new CorDapp JAR file in the `cordapps` folder of all the relevant nodes. You can do this while the nodes are still running.

### 5. Stop the nodes

Have each node operator stop their node. If you are also changing flow definitions, you should perform a `node drain` first to avoid the definition of states or contracts changing whilst a flow is in progress.

### 6. Re-run the network bootstrapper (only if you want to whitelist the new contract)

If you're using the network bootstrapper instead of a network map server and have defined any new contracts, you need to re-run the network bootstrapper to whitelist the new contracts. See [Network Bootstrapper](#).

### 7. Restart the nodes

Have each node operator restart their node.

## 8. Authorise the upgrade

Now that new states and contracts are on the classpath for all the relevant nodes, the nodes must all run the `ContractUpgradeFlow.Authorise` flow. This flow takes a `StateAndRef` of the state to update as well as a reference to the new contract, which must implement the `UpgradedContract` interface.

At any point, a node administrator may de-authorise a contract upgrade by running the `ContractUpgradeFlow.Deauthorise` flow.

## 9. Perform the upgrade

Once all nodes have performed the authorisation process, a **single** node must initiate the upgrade via the `ContractUpgradeFlow.Initiate` flow for each state object. This flow has the following signature:

```
class Initiate<OldState : ContractState, out NewState : ContractState>(
    originalState: StateAndRef<OldState>,
    newContractClass: Class<out UpgradedContract<OldState, NewState>>
) : AbstractStateReplacementFlow.Instigator<OldState, NewState, Class<out_
    ↪UpgradedContract<OldState, NewState>>>(originalState, newContractClass)
```

This flow sub-classes `AbstractStateReplacementFlow`, which can be used to upgrade state objects that do not need a contract upgrade.

Once the flow ends successfully, all the participants of the old state object should have the upgraded state object which references the new contract code.

## 10. Migrate the new upgraded state to the Signature Constraint from the zone constraint

Follow the guide in [API: Contract Constraints](#).

### Points to note

#### Capabilities of the contract upgrade flows

- Despite its name, the `ContractUpgradeFlow` handles the update of both state object definitions and contract logic
- The state can completely change as part of an upgrade! For example, it is possible to transmute a `Cat` state into a `Dog` state, provided that all participants in the `Cat` state agree to the change
- If a node has not yet run the contract upgrade authorisation flow, they will not be able to upgrade the contract and/or state objects
- State schema changes are handled separately

### Logistics

- All nodes need to run the contract upgrade authorisation flow to upgrade the contract and/or state objects
- Only node administrators are able to run the contract upgrade authorisation and deauthorisation flows
- Upgrade authorisations can subsequently be deauthorised

- Only one node should run the contract upgrade initiation flow. If multiple nodes run it for the same StateRef, a double-spend will occur for all but the first completed upgrade
- Upgrades do not have to happen immediately. For a period, the two parties can use the old states and contracts side-by-side
- The supplied upgrade flows upgrade one state object at a time

#### 8.9.4 State schema versioning

By default, all state objects are serialised to the database as a string of bytes and referenced by their StateRef. However, it is also possible to define custom schemas for serialising particular properties or combinations of properties, so that they can be queried from a source other than the Corda Vault. This is done by implementing the `QueryableState` interface and creating a custom object relational mapper for the state. See [API: Persistence](#) for details.

For backwards compatible changes such as adding columns, the procedure for upgrading a state schema is to extend the existing object relational mapper. For example, we can update:

```
object ObligationSchemaV1 : MappedSchema(Obligation::class.java, 1, listOf(ObligationEntity::class.java)) {
    @Entity @Table(name = "obligations")
    class ObligationEntity(obligation: Obligation) : PersistentState() {
        @Column var currency: String = obligation.amount.token.toString()
        @Column var amount: Long = obligation.amount.quantity
        @Column @Lob var lender: ByteArray = obligation.lender.owningKey.encoded
        @Column @Lob var borrower: ByteArray = obligation.borrower.owningKey.encoded
        @Column var linear_id: String = obligation.linearId.id.toString()
    }
}
```

```
public class ObligationSchemaV1 extends MappedSchema {
    public ObligationSchemaV1() {
        super(Obligation.class, 1, ImmutableList.of(ObligationEntity.class));
    }
}

@Entity
@Table(name = "obligations")
public class ObligationEntity extends PersistentState {
    @Column(name = "currency") private String currency;
    @Column(name = "amount") private Long amount;
    @Column(name = "lender") @Lob private byte[] lender;
    @Column(name = "borrower") @Lob private byte[] borrower;
    @Column(name = "linear_id") private UUID linearId;

    protected ObligationEntity() {}

    public ObligationEntity(String currency, Long amount, byte[] lender, byte[]_
        ↪borrower, UUID linearId) {
        this.currency = currency;
        this.amount = amount;
        this.lender = lender;
        this.borrower = borrower;
        this.linearId = linearId;
    }
}
```

(continues on next page)

(continued from previous page)

```

public String getCurrency() {
    return currency;
}

public Long getAmount() {
    return amount;
}

public byte[] getLender() {
    return lender;
}

public byte[] getBorrower() {
    return borrower;
}

public UUID getLinearId() {
    return linearId;
}
}

```

To:

```

object ObligationSchemaV1 : MappedSchema(Obligation::class.java, 1,
    listOf(ObligationEntity::class.java)) {
    @Entity @Table(name = "obligations")
    class ObligationEntity(obligation: Obligation) : PersistentState() {
        @Column var currency: String = obligation.amount.token.toString()
        @Column var amount: Long = obligation.amount.quantity
        @Column @Lob var lender: ByteArray = obligation.lender.owningKey.encoded
        @Column @Lob var borrower: ByteArray = obligation.borrower.owningKey.encoded
        @Column var linear_id: String = obligation.linearId.id.toString()
        @Column var defaulted: Bool = obligation.amount.inDefault
        // NEW COLUMN!
    }
}

```

```

public class ObligationSchemaV1 extends MappedSchema {
    public ObligationSchemaV1() {
        super(Obligation.class, 1, ImmutableList.of(ObligationEntity.class));
    }
}

@Entity
@Table(name = "obligations")
public class ObligationEntity extends PersistentState {
    @Column(name = "currency") private String currency;
    @Column(name = "amount") private Long amount;
    @Column(name = "lender") @Lob private byte[] lender;
    @Column(name = "borrower") @Lob private byte[] borrower;
    @Column(name = "linear_id") private UUID linearId;
    @Column(name = "defaulted") private Boolean defaulted;
    // NEW COLUMN!

    protected ObligationEntity() {}

    public ObligationEntity(String currency, Long amount, byte[] lender, byte[]_
        borrower, UUID linearId, Boolean defaulted) {
}

```

(continues on next page)

(continued from previous page)

```

    this.currency = currency;
    this.amount = amount;
    this.lender = lender;
    this.borrower = borrower;
    this.linearId = linearId;
    this.defaulted = defaulted;
}

public String getCurrency() {
    return currency;
}

public Long getAmount() {
    return amount;
}

public byte[] getLender() {
    return lender;
}

public byte[] getBorrower() {
    return borrower;
}

public UUID getLinearId() {
    return linearId;
}

public Boolean isDefaulted() {
    return defaulted;
}
}
}

```

Thus adding a new column with a default value.

To make a non-backwards compatible change, the `ContractUpgradeFlow` or `AbstractStateReplacementFlow` must be used, as changes to the state are required. To make a backwards-incompatible change such as deleting a column (e.g. because a property was removed from a state object), the procedure is to define another object relational mapper, then add it to the `supportedSchemas` property of your `QueryableState`, like so:

```

override fun supportedSchemas(): Iterable<MappedSchema> = listOf(ExampleSchemaV1,
    ExampleSchemaV2)

```

```

@Override public Iterable<MappedSchema> supportedSchemas() {
    return ImmutableList.of(new ExampleSchemaV1(), new ExampleSchemaV2());
}

```

Then, in `generateMappedObject`, add support for the new schema:

```

override fun generateMappedObject(schema: MappedSchema): PersistentState {
    return when (schema) {
        is DummyLinearStateSchemaV1 -> // Omitted.
        is DummyLinearStateSchemaV2 -> // Omitted.
        else -> throw IllegalArgumentException("Unrecognised schema $schema")
    }
}

```

```

@Override public PersistentState generateMappedObject(MappedSchema schema) {
    if (schema instanceof DummyLinearStateSchemaV1) {
        // Omitted.
    } else if (schema instanceof DummyLinearStateSchemaV2) {
        // Omitted.
    } else {
        throw new IllegalArgumentException("Unrecognised schema $schema");
    }
}

```

With this approach, whenever the state object is stored in the vault, a representation of it will be stored in two separate database tables where possible - one for each supported schema.

## 8.9.5 Serialisation

### The Corda serialisation format

Currently, the serialisation format for everything except flow checkpoints (which uses a Kryo-based format) is based on AMQP 1.0, a self-describing and controllable serialisation format. AMQP is desirable because it allows us to have a schema describing what has been serialized alongside the data itself. This assists with versioning and deserialising long-ago archived data, among other things.

### Writing classes that meet the serialisation format requirements

Although not strictly related to versioning, AMQP serialisation dictates that we must write our classes in a particular way:

- Your class must have a constructor that takes all the properties that you wish to record in the serialized form. This is required in order for the serialization framework to reconstruct an instance of your class
- If more than one constructor is provided, the serialization framework needs to know which one to use. The `@ConstructorForDeserialization` annotation can be used to indicate the chosen constructor. For a Kotlin class without the `@ConstructorForDeserialization` annotation, the primary constructor is selected
- The class must be compiled with parameter names in the .class file. This is the default in Kotlin but must be turned on in Java (using the `-parameters` command line option to `javac`)
- Your class must provide a Java Bean getter for each of the properties in the constructor, with a matching name. For example, if a class has the constructor parameter `foo`, there must be a getter called `getFoo()`. If `foo` is a boolean, the getter may optionally be called `isFoo()`. This is why the class must be compiled with parameter names turned on
- The class must be annotated with `@CordaSerializable`
- The declared types of constructor arguments/getters must be supported, and where generics are used the generic parameter must be a supported type, an open wildcard (\*), or a bounded wildcard which is currently widened to an open wildcard
- Any superclass must adhere to the same rules, but can be abstract
- Object graph cycles are not supported, so an object cannot refer to itself, directly or indirectly

## 8.9.6 Testing CorDapp upgrades

At the time of this writing there is no platform support to test CorDapp upgrades. There are plans to add support in a future version. This means that it is not possible to write automated tests using just the provided tooling.

To test an implicit upgrade, you must simulate a network that initially has only nodes with the old version of the CorDapp, and then nodes gradually transition to the new version. Typically, in such a complex upgrade scenario, there must be a deadline by which time all nodes that want to continue to use the CorDapp must upgrade. To achieve this, this deadline must be configured in the flow logic which must only use new features afterwards.

This can be simulated with a scenario like this:

1. Write and individually test the new version of the state and contract.
2. Setup a network of nodes with the previous version. In the simplest form, `deployNodes` can be used for this purpose.
3. Run some transactions between nodes.
4. Upgrade a couple of nodes to the new version of the CorDapp.
5. Continue running transactions between various combinations of versions. Also make sure transactions that were created between nodes with the new version are being successfully read by nodes with the old CorDapp.
6. Upgrade all nodes and simulate the deadline expiration.
7. Make sure old transactions can be consumed, and new features are successfully used in new transactions.

## 8.10 CorDapp constraints migration

---

**Note:** Before reading this page, you should be familiar with the key concepts of [Contract Constraints](#).

---

Corda 4 introduces and recommends building signed CorDapps that issue states with signature constraints. Existing on ledger states issued before Corda 4 are not automatically transitioned to new signature constraints when building transactions in Corda 4. This document explains how to modify existing CorDapp flows to explicitly consume and evolve pre Corda 4 states, and outlines a future mechanism where such states will transition automatically (without explicit migration code).

Faced with the exercise of upgrading an existing Corda 3.x CorDapp to Corda 4, you need to consider the following:

- What existing unconsumed states have been issued on ledger by a previous version of this CorDapp and using other constraint types?

If you have existing **hash** constrained states see [Migrating hash constraints](#).

If you have existing **CZ whitelisted** constrained states see [Migrating CZ whitelisted constraints](#).

If you have existing **always accept** constrained states these are not consumable nor evolvable as they offer no security and should only be used in test environments.

- What type of contract states does my CorDapp use?

**Linear states** typically evolve over an extended period of time (defined by the lifecycle of the associated business use case), and thus are prime candidates for constraints migration.

**Fungible states** are created by an issuer and transferred around a Corda network until explicitly exited (by the same issuer). They do not evolve as linear states, but are transferred between participants on a network. Their consumption may produce additional new output states to represent adjustments to the original state (e.g. change when spending cash). For the purposes of constraints migration, it is desirable that any new output states are produced using the new Corda 4 signature constraint types.

Where you have long transaction chains of fungible states, it may be advisable to send them back to the issuer for re-issuance (this is called “chain snipping” and has performance advantages as well as simplifying constraints type migration).

- Should I use the **implicit** or **explicit** upgrade path?

The general recommendation for Corda 4 is to use **implicit** upgrades for the reasons described [here](#).

**Implicit** upgrades allow pre-authorising multiple implementations of the contract ahead of time. They do not require additional coding and do not incur a complex choreographed operational upgrade process.

**Warning:** The steps outlined in this page assume you are using the same CorDapp Contract (eg. same state definition, commands and verification code) and wish to use that CorDapp to leverage the upgradeability benefits of Corda 4 signature constraints. If you are looking to upgrade code within an existing Contract CorDapp please read [Contract and state versioning](#) and [CorDapp Upgradeability Guarantees](#) to understand your options.

Please also remember that *states are always consumable if the version of the CorDapp that issued (created) them is installed*. In the simplest of scenarios it may be easier to re-issue existing hash or CZ whitelist constrained states (eg. exit them from the ledger using the original unsigned CorDapp and re-issuing them using the new signed CorDapp).

### 8.10.1 Hash constraints migration

---

**Note:** These instructions only apply to CorDapp Contract JARs (unless otherwise stated).

---

#### Corda 4.0

Corda 4.0 requires some additional steps to consume and evolve pre-existing on-ledger **hash** constrained states:

1. All Corda Nodes in the same CZ or business network that may encounter a transaction chain with a hash constrained state must be started using relaxed hash constraint checking mode as described in [Hash constrained states in private networks](#).
2. CorDapp flows that build transactions using pre-existing *hash-constrained* states must explicitly set output states to use *signature constraints* and specify the related public key(s) used in signing the associated CorDapp Contract JAR:

```
// This will read the signers for the deployed CorDapp.
val attachment = this.serviceHub.cordappProvider.
    ↪getContractAttachmentID(contractClass)
val signers = this.serviceHub.attachments.openAttachment(attachment!!)!!!.signerKeys

// Create the key that will have to pass for all future versions.
val ownersKey = signers.first()

val txBuilder = TransactionBuilder(notary)
    // Set the Signature constraint on the new state to migrate away from the
    ↪hash constraint.
    .addOutputState(outputState, constraint =
        ↪SignatureAttachmentConstraint(ownersKey))
```

```
// This will read the signers for the deployed CorDapp.
SecureHash attachment = this.getServiceHub().getCordappProvider() .
    ↪getContractAttachmentID(contractClass);
List<PublicKey> signers = this.getServiceHub().getAttachments() .
    ↪openAttachment(attachment).getSignerKeys();

// Create the key that will have to pass for all future versions.
PublicKey ownersKey = signers.get(0);

TransactionBuilder txBuilder = new TransactionBuilder(notary)
    // Set the Signature constraint on the new state to migrate away from the
    ↪hash constraint.
    .addOutputState(outputState, myContract, new
    ↪SignatureAttachmentConstraint(ownersKey))
```

3. As a node operator you need to add the new signed version of the contracts CorDapp to the /cordapps folder together with the latest version of the flows jar. Please also ensure that the original unsigned contracts CorDapp is removed from the /cordapps folder (this will already be present in the nodes attachments store) to ensure the lookup code in step 2 retrieves the correct signed contract CorDapp JAR.

### Later releases

The next version of Corda will provide automatic transition of *hash constrained* states. This means that signed CorDapps running on a Corda 4.x node will automatically propagate any pre-existing on-ledger *hash-constrained* states (and generate *signature-constrained* outputs) when the system property to break constraints is set.

### 8.10.2 CZ whitelisted constraints migration

---

**Note:** These instructions only apply to CorDapp Contract JARs (unless otherwise stated).

---

#### Corda 4.0

Corda 4.0 requires some additional steps to consume and evolve pre-existing on-ledger **CZ whitelisted** constrained states:

1. As the original developer of the CorDapp, the first step is to sign the latest version of the JAR that was released (see *Building and installing a CorDapp*). The key used for signing will be used to sign all subsequent releases, so it should be stored appropriately. The JAR can be signed by multiple keys owned by different parties and it will be expressed as a CompositeKey in the SignatureAttachmentConstraint (See *API: Core types*).
2. The new Corda 4 signed CorDapp JAR must be registered with the CZ network operator (as whitelisted in the network parameters which are distributed to all nodes in that CZ). The CZ network operator should check that the JAR is signed and not allow any more versions of it to be whitelisted in the future. From now on the development organisation that signed the JAR is responsible for signing new versions.

The process of CZ network CorDapp whitelisting depends on how the Corda network is configured:

- if using a hosted CZ network (such as *The Corda Network* or *UAT Environment*) running an Identity Operator (formerly known as Doorman) and Network Map Service, you should manually send the hashes of the two JARs to the CZ network operator and request these be added using their network parameter update process.

- if using a local network created using the Network Bootstrapper tool, please follow the instructions in [Updating the contract whitelist for bootstrapped networks](#) to can add both CorDapp Contract JAR hashes.
3. Any flows that build transactions using this CorDapp will have the responsibility of transitioning states to the `SignatureAttachmentConstraint`. This is done explicitly in the code by setting the constraint of the output states to signers of the latest version of the whitelisted jar:

```
// This will read the signers for the deployed CorDapp.
val attachment = this.serviceHub.cordappProvider.
    ↪getContractAttachmentID(contractClass)
val signers = this.serviceHub.attachments.openAttachment(attachment!!)!!.signerKeys

// Create the key that will have to pass for all future versions.
val ownersKey = signers.first()

val txBuilder = TransactionBuilder(notary)
    // Set the Signature constraint on the new state to migrate away from the
    ↪WhitelistConstraint.
    .addOutputState(outputState, constraint =
    ↪SignatureAttachmentConstraint(ownersKey))
```

```
// This will read the signers for the deployed CorDapp.
SecureHash attachment = this.getServiceHub().getCordappProvider().
    ↪getContractAttachmentID(contractClass);
List<PublicKey> signers = this.getServiceHub().getAttachments().
    ↪openAttachment(attachment).getSignerKeys();

// Create the key that will have to pass for all future versions.
PublicKey ownersKey = signers.get(0);

TransactionBuilder txBuilder = new TransactionBuilder(notary)
    // Set the Signature constraint on the new state to migrate away from the
    ↪WhitelistConstraint.
    .addOutputState(outputState, myContract, new
    ↪SignatureAttachmentConstraint(ownersKey))
```

4. As a node operator you need to add the new signed version of the contracts CorDapp to the `/cordapps` folder together with the latest version of the flows jar. Please also ensure that the original unsigned contracts CorDapp is removed from the `/cordapps` folder (this will already be present in the nodes attachments store) to ensure the lookup code in step 3 retrieves the correct signed contract CorDapp JAR.

## Later releases

The next version of Corda will provide automatic transition of *CZ whitelisted* constrained states. This means that signed CorDapps running on a Corda 4.x node will automatically propagate any pre-existing on-ledger *CZ whitelisted* constrained states (and generate *signature* constrained outputs).

## 8.11 CorDapp Upgradeability Guarantees

### 8.11.1 Corda 4.0

Corda 4 introduces a number of advanced features (such as signature constraints), and data security model improvements (such as attachments trust checking and classloader isolation of contract attachments for transaction building and verification).

The following guarantees are made for CorDapps running on Corda 4.0

- Compliant CorDapps compiled with previous versions of Corda (from 3.0) will execute without change on Corda 4.0

---

**Note:** by “compliant”, we mean CorDapps that do not utilise Corda internal, non-stable or other non-committed public Corda APIs.

---

Recommendation: security hardening changes in flow processing, specifically the `FinalityFlow`, recommend upgrading existing CorDapp receiver flows to use the new APIs and thus opting in to platform version 4. See [Step 5. Security: Upgrade your use of FinalityFlow](#) for more information.

- All constraint types (hash, CZ whitelisted, signature) are consumable within the same transaction if there is an associated contract attachment that satisfies all of them.
- CorDapp Contract states generated on ledger using hash constraints are not directly migratable to signature constraints in this release. Your compatibility zone operator may whitelist a JAR previously used to issue hash constrained states, and then you can follow the manual process described in the paragraph below to migrate these to signature constraints. See [CorDapp constraints migration](#) for more information.
- CorDapp Contract states generated on ledger using CZ whitelisted constraints are migratable to signature constraints using a manual process that requires programmatic code changes. See [CZ whitelisted constraints migration](#) for more information.
- Explicit Contract Upgrades are only supported for hash and CZ whitelisted constraint types. See [Performing explicit contract and state upgrades](#) for more information.
- CorDapp contract attachments are not trusted from remote peers over the p2p network for the purpose of transaction verification. A node operator must locally install *all* versions of a Contract attachment to be able to resolve a chain of contract states from its original version. The RPC `uploadAttachment` mechanism can be used to achieve this (as well as conventional loading of a CorDapp by installing it in the nodes /cordapp directory). See [Installing the CorDapp JAR](#) and [CorDapp Contract Attachments](#) for more information.
- CorDapp contract attachment classloader isolation has some important side-effects and edge cases to consider:
  1. Contract attachments should include all 3rd party library dependencies in the same packaged JAR - we call this a “Fat JAR”, meaning that all dependencies are resolvable by the classloader by only loading a single JAR.
  2. Contract attachments that depend on other Contract attachments from a different packaged JAR are currently supported in so far as the Attachments Classloader will attempt to resolve any external dependencies from the node’s application classloader. It is thus paramount that dependent Contract Attachments are loaded upon node startup from the respective /cordapps directory.
- Rolling upgrades are partially supported. A Node operator may choose to manually upload (via the RPC attachments uploader mechanism) a later version of a Contract Attachment than the version their node is currently using for the purposes of transaction verification (received from remote peers). However, they will only be able to build new transactions with the version that is currently loaded (installed from the nodes /cordapps directory) in their node.
- Finance CorDapp (v4) Whilst experimental, our test coverage has confirmed that states generated with the Finance CorDapp are interchangeable across Open Source and Enterprise distributions. This has been made possible by releasing a single 4.0 version of the Finance Contracts CorDapp. Please note the Finance application will be superseded shortly by the new Tokens SDK (<https://github.com/corda/token-sdk>)

### 8.11.2 Later releases

The following additional capabilities are under consideration for delivery in follow-up releases to Corda 4.0:

- CorDapp Contract states generated on ledger using hash constraints will be automatically migrated to signature constraints when building new transactions where the latest installed contract Jar is signed as per [CorDapp Jar signing](#).
- CorDapp Contract states generated on ledger using CZ whitelisted constraints will be automatically migrated to signature constraints when building new transactions where the latest installed contract Jar is signed as per [CorDapp Jar signing](#).
- Explicit Contract Upgrades will be supported for all constraint types: hash, CZ whitelisted and signature. In practice, it should only be necessary to upgrade from hash or CZ whitelisted to new signature constrained contract types. signature constrained Contracts are upgradeable seamlessly (through built in serialization and code signing controls) without requiring explicit upgrades.
- Contract attachments will be able to explicitly declare their dependencies on other Contract attachments such that these are automatically loaded by the Attachments Classloader (rendering the 4.0 fallback to application classloader mechanism redundant). This improved modularity removes the need to “Fat JAR” all dependencies together in a single jar.
- Rolling upgrades will be fully supported. A Node operator will be able to pre-register (by hash or code signing public key) versions of CorDapps they are not yet ready to install locally, but wish to use for the purposes of transaction verification with peers running later versions of a CorDapp.

---

**Note:** Trusted downloading and execution of contract attachments from remote peers will not be integrated until secure JVM sand-boxing is available.

---

## 8.12 Secure coding guidelines

The platform does what it can to be secure by default and safe by design. Unfortunately the platform cannot prevent every kind of security mistake. This document describes what to think about when writing applications to block various kinds of attack. Whilst it may be tempting to just assume no reasonable counterparty would attempt to subvert your trades using flow level attacks, relying on trust for software security makes it harder to scale up your operations later when you might want to add counterparties quickly and without extensive vetting.

### 8.12.1 Flows

*Writing flows* are how your app communicates with other parties on the network. Therefore they are the typical entry point for malicious data into your app and must be treated with care.

The `receive` methods return data wrapped in the `UntrustworthyData<T>` marker type. This type doesn't add any functionality, it's only there to remind you to properly validate everything that you get from the network. Remember that the other side may *not* be running the code you provide to take part in the flow: they are allowed to do anything! Things to watch out for:

- A transaction that doesn't match a partial transaction built or proposed earlier in the flow, for instance, if you propose to trade a cash state worth \$100 for an asset, and the transaction to sign comes back from the other side, you must check that it points to the state you actually requested. Otherwise the attacker could get you to sign a transaction that spends a much larger state to you, if they know the ID of one!
- A transaction that isn't of the right type. There are two transaction types: general and notary change. If you are expecting one type but get the other you may find yourself signing a transaction that transfers your assets to the control of a hostile notary.
- Unexpected changes in any part of the states in a transaction. If you have access to all the needed data, you could re-run the builder logic and do a comparison of the resulting states to ensure that it's what you expected.

For instance if the data needed to construct the next state is available to both parties, the function to calculate the transaction you want to mutually agree could be shared between both classes implementing both sides of the flow.

The theme should be clear: signing is a very sensitive operation, so you need to be sure you know what it is you are about to sign, and that nothing has changed in the small print! Once you have provided your signature over a transaction to a counterparty, there is no longer anything you can do to prevent them from committing it to the ledger.

## 8.12.2 Contracts

Contracts are arbitrary functions inside a JVM sandbox and therefore they have a lot of leeway to shoot themselves in the foot. Things to watch out for:

- Changes in states that should not be allowed by the current state transition. You will want to check that no fields are changing except the intended fields!
- Accidentally catching and discarding exceptions that might be thrown by validation logic.
- Calling into other contracts via virtual methods if you don't know what those other contracts are or might do.

## 8.13 Configuring Responder Flows

A flow can be a fairly complex thing that interacts with many backend systems, and so it is likely that different users of a specific CordApp will require differences in how flows interact with their specific infrastructure.

Corda supports this functionality by providing two mechanisms to modify the behaviour of apps in your node.

### 8.13.1 Subclassing a Flow

If you have a workflow which is mostly common, but also requires slight alterations in specific situations, most developers would be familiar with refactoring into *Base* and *Sub* classes. A simple example is shown below.

```
@InitiatedBy(Initiator::class)
open class BaseResponder(internal val otherSideSession: FlowSession) : FlowLogic<Unit>
    () {
    @Suspendable
    override fun call() {
        otherSideSession.send(getMessage())
    }
    protected open fun getMessage() = "This Is the Legacy Responder"
}

@InitiatedBy(Initiator::class)
class SubResponder(otherSideSession: FlowSession) : BaseResponder(otherSideSession) {
    override fun getMessage(): String {
        return "This is the sub responder"
    }
}
```

```
@InitiatingFlow
public class Initiator extends FlowLogic<String> {
    private final Party otherSide;

    public Initiator(Party otherSide) {
```

(continues on next page)

(continued from previous page)

```

        this.otherSide = otherSide;
    }

    @Override
    public String call() throws FlowException {
        return initiateFlow(otherSide).receive(String.class).unwrap(it) -> it;
    }
}

@InitiatedBy(Initiator.class)
public class BaseResponder extends FlowLogic<Void> {
    private FlowSession counterpartySession;

    public BaseResponder(FlowSession counterpartySession) {
        super();
        this.counterpartySession = counterpartySession;
    }

    @Override
    public Void call() throws FlowException {
        counterpartySession.send(getMessage());
        return Void;
    }

    protected String getMessage() {
        return "This Is the Legacy Responder";
    }
}

@InitiatedBy(Initiator.class)
public class SubResponder extends BaseResponder {
    public SubResponder(FlowSession counterpartySession) {
        super(counterpartySession);
    }

    @Override
    protected String getMessage() {
        return "This is the sub responder";
    }
}

```

Corda would detect that both `BaseResponder` and `SubResponder` are configured for responding to `Initiator`. Corda will then calculate the hops to `FlowLogic` and select the implementation which is furthest distance, ie: the most subclassed implementation. In the above example, `SubResponder` would be selected as the default responder for `Initiator`.

---

**Note:** The flows do not need to be within the same CordApp, or package, therefore to customise a shared app you obtained from a third party, you'd write your own CorDapp that subclasses the first.

---

### 8.13.2 Overriding a flow via node configuration

Whilst the subclassing approach is likely to be useful for most applications, there is another mechanism to override this behaviour. This would be useful if for example, a specific CordApp user requires such a different responder that

subclassing an existing flow would not be a good solution. In this case, it's possible to specify a hardcoded flow via the node configuration.

---

**Note:** A new responder written to override an existing responder must still be annotated with `@InitiatedBy` referencing the base initiator.

---

The configuration section is named `flowOverrides` and it accepts an array of `overrides`

```
flowOverrides {
    overrides=[
        {
            initiator="net.corda.Initiator"
            responder="net.corda.BaseResponder"
        }
    ]
}
```

The cordform plugin also provides a `flowOverride` method within the `deployNodes` block which can be used to override a flow. In the below example, we will override the `SubResponder` with `BaseResponder`

```
node {
    name "O=Bank,L=London,C=GB"
    p2pPort 10025
    rpcUsers = ext.rpcUsers
    rpcSettings {
        address "localhost:10026"
        adminAddress "localhost:10027"
    }
    extraConfig = ['h2Settings.address' : 'localhost:10035']
    flowOverride("net.corda.Initiator", "net.corda.BaseResponder")
}
```

This will generate the corresponding `flowOverrides` section and place it in the configuration for that node.

### 8.13.3 Modifying the behaviour of `@InitiatingFlow(s)`

It is likely that initiating flows will also require changes to reflect the different systems that are likely to be encountered. At the moment, corda provides the ability to subclass an Initiator, and ensures that the correct responder will be invoked. In the below example, we will change the behaviour of an Initiator from filtering Notaries out from comms, to only communicating with Notaries

```
@InitiatingFlow
@StartableByRPC
@StartableByService
open class BaseInitiator : FlowLogic<String>() {
    @Suspendable
    override fun call(): String {
        val partiesToTalkTo = serviceHub.networkMapCache.allNodes
            .filterNot { it.legalIdentities.first() in serviceHub.
        ↵networkMapCache.notaryIdentities }
            .filterNot { it.legalIdentities.first().name == ourIdentity.
        ↵name }.map { it.legalIdentities.first() }
        val responses = ArrayList<String>()
        for (party in partiesToTalkTo) {
            val session = initiateFlow(party)
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    val received = session.receive<String>().unwrap { it }
    responses.add(party.name.toString() + " responded with backend:
    ↵" + received)
}
return "${getFlowName()} received the following \n" + responses.
joinToString("\n") { it }
}

open fun getFlowName(): String {
    return "Normal Computer"
}

@StartableByRPC
@StartableByService
class NotaryOnlyInitiator : BaseInitiator() {
    @Suspendable
    override fun call(): String {
        return "Notary Communicator received:\n" + serviceHub.
        ↵networkMapCache.notaryIdentities.map {
            "Notary: ${it.name.organisation} is using a " + initiateFlow(it).
        ↵receive<String>().unwrap { it }
            }.joinToString("\n") { it }
    }
}

```

**Warning:** The subclass must not have the @InitiatingFlow annotation.

Corda will use the first annotation detected in the class hierarchy to determine which responder should be invoked. So for a Responder similar to

```

@InitiatedBy(BaseInitiator::class)
class BobbyResponder(othersideSession: FlowSession) :_
    BaseResponder(othersideSession) {
    override fun getMessageFromBackend(): String {
        return "Robert'; DROP TABLE STATES;"
    }
}

```

it would be possible to invoke either BaseInitiator or NotaryOnlyInitiator and BobbyResponder would be used to reply.

**Warning:** You must ensure the sequence of sends/receives/subFlows in a subclass are compatible with the parent.

## 8.14 Flow cookbook

This flow showcases how to use Corda's API, in both Java and Kotlin.

```

@file:Suppress("UNUSED_VARIABLE", "unused", "DEPRECATION")

package net.corda.docs.kotlin

```

(continues on next page)

(continued from previous page)

```

import co.paralleluniverse.fibers.Suspendable
import net.corda.core.contracts.*
import net.corda.core.crypto.SecureHash
import net.corda.core.crypto.TransactionSignature
import net.corda.core.crypto.generateKeyPair
import net.corda.core.flows.*
import net.corda.core.identity.CordaX500Name
import net.corda.core.identity.Party
import net.corda.core.identity.PartyAndCertificate
import net.corda.core.internal.FetchDataFlow
import net.corda.core.node.services.Vault.Page
import net.corda.core.node.services.queryBy
import net.corda.core.node.services.vault.QueryCriteria.VaultQueryCriteria
import net.corda.core.transactions.LedgerTransaction
import net.corda.core.transactions.SignedTransaction
import net.corda.core.transactions.TransactionBuilder
import net.corda.core.utilities.ProgressTracker
import net.corda.core.utilities.ProgressTracker.Step
import net.corda.core.utilities.UntrustworthyData
import net.corda.core.utilities.seconds
import net.corda.core.utilities.unwrap
import net.corda.finance.contracts.asset.Cash
import net.corda.testing.contracts.DummyContract
import net.corda.testing.contracts.DummyState
import java.security.PublicKey
import java.security.Signature
import java.time.Instant

// ``InitiatorFlow`` is our first flow, and will communicate with
// ``ResponderFlow``, below.
// We mark ``InitiatorFlow`` as an ``InitiatingFlow``, allowing it to be
// started directly by the node.
@InitiatingFlow
// We also mark ``InitiatorFlow`` as ``StartableByRPC``, allowing the
// node's owner to start the flow via RPC.
@StartableByRPC
// Every flow must subclass ``FlowLogic``. The generic indicates the
// flow's return type.
class InitiatorFlow(val arg1: Boolean, val arg2: Int, private val counterparty: Party,
    ↪ val regulator: Party) : FlowLogic<Unit>() {

    /**
     * WIRING UP THE PROGRESS TRACKER *
     */
    // Giving our flow a progress tracker allows us to see the flow's
    // progress visually in our node's CRaSH shell.
    // DOCSTART 17
    companion object {
        object ID_OTHER_NODES : Step("Identifying other nodes on the network.")
        object SENDING_AND RECEIVING_DATA : Step("Sending data between parties.")
        object EXTRACTING_VAULT_STATES : Step("Extracting states from the vault.")
        object OTHER_TX_COMPONENTS : Step("Gathering a transaction's other components.
    ↪ ")
        object TX_BUILDING : Step("Building a transaction.")
        object TX_SIGNING : Step("Signing a transaction.")
        object TX_VERIFICATION : Step("Verifying a transaction.")
    }
}

```

(continues on next page)

(continued from previous page)

```

object SIGS_GATHERING : Step("Gathering a transaction's signatures.") {
    // Wiring up a child progress tracker allows us to see the
    // subflow's progress steps in our flow's progress tracker.
    override fun childProgressTracker() = CollectSignaturesFlow.tracker()
}

object VERIFYING_SIGS : Step("Verifying a transaction's signatures.")
object FINALISATION : Step("Finalising a transaction.") {
    override fun childProgressTracker() = FinalityFlow.tracker()
}

fun tracker() = ProgressTracker(
    ID_OTHER_NODES,
    SENDING_AND RECEIVING_DATA,
    EXTRACTING_VAULT_STATES,
    OTHER_TX_COMPONENTS,
    TX_BUILDING,
    TX_SIGNING,
    TX_VERIFICATION,
    SIGS_GATHERING,
    VERIFYING_SIGS,
    FINALISATION
)
}
// DOCEND 17

override val progressTracker: ProgressTracker = tracker()

@Suppress("RemoveExplicitTypeArguments")
@suspendable
override fun call() {
    // We'll be using a dummy public key for demonstration purposes.
    val dummyPubKey: PublicKey = generateKeyPair().public

    /*-----
     * IDENTIFYING OTHER NODES *
    -----*/
// DOCSTART 18
progressTracker.currentStep = ID_OTHER_NODES
// DOCEND 18

// A transaction generally needs a notary:
// - To prevent double-spends if the transaction has inputs
// - To serve as a timestamping authority if the transaction has a
// time-window
// We retrieve the notary from the network map.
// DOCSTART 01
val notaryName: CordaX500Name = CordaX500Name(
    organisation = "Notary Service",
    locality = "London",
    country = "GB")
val specificNotary: Party = serviceHub.networkMapCache.getNotary(notaryName) !!
// Alternatively, we can pick an arbitrary notary from the notary
// list. However, it is always preferable to specify the notary
// explicitly, as the notary list might change when new notaries are
// introduced, or old ones decommissioned.
val firstNotary: Party = serviceHub.networkMapCache.notaryIdentities.first()

```

(continues on next page)

(continued from previous page)

```

// DOCEND 01

// We may also need to identify a specific counterparty. We do so
// using the identity service.
// DOCSTART 02
val counterpartyName: CordaX500Name = CordaX500Name(
    organisation = "NodeA",
    locality = "London",
    country = "GB")
val namedCounterparty: Party = serviceHub.identityService.
    ↪wellKnownPartyFromX500Name(counterpartyName) ?:
        throw IllegalArgumentException("Couldn't find counterparty for NodeA_"
    ↪in identity service")
val keyedCounterparty: Party = serviceHub.identityService.
    ↪partyFromKey(dummyPubKey) ?:
        throw IllegalArgumentException("Couldn't find counterparty with key: "
    ↪$dummyPubKey in identity service")
    // DOCEND 02

/*-----
 * SENDING AND RECEIVING DATA *
-----*/
progressTracker.currentStep = SENDING_AND_RECEIVING_DATA

// We start by initiating a flow session with the counterparty. We
// will use this session to send and receive messages from the
// counterparty.
// DOCSTART initiateFlow
val counterpartySession: FlowSession = initiateFlow(counterparty)
// DOCEND initiateFlow

// We can send arbitrary data to a counterparty.
// If this is the first ``send``, the counterparty will either:
// 1. Ignore the message if they are not registered to respond
// to messages from this flow.
// 2. Start the flow they have registered to respond to this flow,
// and run the flow until the first call to ``receive``, at
// which point they process the message.
// In other words, we are assuming that the counterparty is
// registered to respond to this flow, and has a corresponding
// ``receive`` call.
// DOCSTART 04
counterpartySession.send(Any())
// DOCEND 04

// We can wait to receive arbitrary data of a specific type from a
// counterparty. Again, this implies a corresponding ``send`` call
// in the counterparty's flow. A few scenarios:
// - We never receive a message back. In the current design, the
// flow is paused until the node's owner kills the flow.
// - Instead of sending a message back, the counterparty throws a
// ``FlowException``. This exception is propagated back to us,
// and we can use the error message to establish what happened.
// - We receive a message back, but it's of the wrong type. In
// this case, a ``FlowException`` is thrown.
// - We receive back a message of the correct type. All is good.
// 
```

(continues on next page)

(continued from previous page)

```

// Upon calling ``receive()`` (or ``sendAndReceive()``), the
// ``FlowLogic`` is suspended until it receives a response.
//
// We receive the data wrapped in an ``UntrustworthyData``
// instance. This is a reminder that the data we receive may not
// be what it appears to be! We must unwrap the
// ``UntrustworthyData`` using a lambda.
// DOCSTART 05
val packet1: UntrustworthyData<Int> = counterpartySession.receive<Int>()
val int: Int = packet1.unwrap { data ->
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    data
}
// DOCEND 05

// We can also use a single call to send data to a counterparty
// and wait to receive data of a specific type back. The type of
// data sent doesn't need to match the type of the data received
// back.
// DOCSTART 07
val packet2: UntrustworthyData<Boolean> = counterpartySession.sendAndReceive
→<Boolean>("You can send and receive any class!")
val boolean: Boolean = packet2.unwrap { data ->
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    data
}
// DOCEND 07

// We're not limited to sending to and receiving from a single
// counterparty. A flow can send messages to as many parties as it
// likes, and each party can invoke a different response flow.
// DOCSTART 06
val regulatorSession: FlowSession = initiateFlow(regulator)
regulatorSession.send()
val packet3: UntrustworthyData<Any> = regulatorSession.receive<Any>()
// DOCEND 06

// We may also batch receives in order to increase performance. This
// ensures that only a single checkpoint is created for all received
// messages.
// Type-safe variant:
val signatures: List<UntrustworthyData<Signature>> =
    receiveAll(Signature::class.java, listOf(counterpartySession,
→regulatorSession))
// Dynamic variant:
val messages: Map<FlowSession, UntrustworthyData<*>> =
    receiveAllMap(mapOf(
        counterpartySession to Boolean::class.java,
        regulatorSession to String::class.java
    ))
/*
-----
* EXTRACTING STATES FROM THE VAULT *

```

(continues on next page)

(continued from previous page)

```

-----*/
progressTracker.currentStep = EXTRACTING_VAULT_STATES

// Let's assume there are already some ``DummyState``'s in our
// node's vault, stored there as a result of running past flows,
// and we want to consume them in a transaction. There are many
// ways to extract these states from our vault.

// For example, we would extract any unconsumed ``DummyState``'s
// from our vault as follows:
val criteria: VaultQueryCriteria = VaultQueryCriteria() // default is
→UNCONSUMED
val results: Page<DummyState> = serviceHub.vaultService.queryBy<DummyState>
→(criteria)
val dummyStates: List<StateAndRef<DummyState>> = results.states

// For a full list of the available ways of extracting states from
// the vault, see the Vault Query docs page.

// When building a transaction, input states are passed in as
// ``StateRef`` instances, which pair the hash of the transaction
// that generated the state with the state's index in the outputs
// of that transaction. In practice, we'd pass the transaction hash
// or the ``StateRef`` as a parameter to the flow, or extract the
// ``StateRef`` from our vault.
// DOCSTART 20
val ourStateRef: StateRef = StateRef(SecureHash.sha256("DummyTransactionHash
→"), 0)
// DOCEND 20
// A ``StateAndRef`` pairs a ``StateRef`` with the state it points to.
// DOCSTART 21
val ourStateAndRef: StateAndRef<DummyState> = serviceHub.toStateAndRef
→<DummyState>(ourStateRef)
// DOCEND 21

/*-----
 * GATHERING OTHER TRANSACTION COMPONENTS *
-----*/
progressTracker.currentStep = OTHER_TX_COMPONENTS

// Reference input states are constructed from StateAndRefs.
// DOCSTART 55
val referenceState: ReferencedStateAndRef<DummyState> = ourStateAndRef.
→referenced()
// DOCEND 55
// Output states are constructed from scratch.
// DOCSTART 22
val ourOutputState: DummyState = DummyState()
// DOCEND 22
// Or as copies of other states with some properties changed.
// DOCSTART 23
val ourOtherOutputState: DummyState = ourOutputState.copy(magicNumber = 77)
// DOCEND 23

// We then need to pair our output state with a contract.
// DOCSTART 47
val ourOutput: StateAndContract = StateAndContract(ourOutputState,
→DummyContract.PROGRAM_ID)

```

(continues on next page)

(continued from previous page)

```

// DOCEND 47

    // Commands pair a ``CommandData`` instance with a list of
    // public keys. To be valid, the transaction requires a signature
    // matching every public key in all of the transaction's commands.
    // DOCSTART 24
    val commandData: DummyContract.Commands.Create = DummyContract.Commands.
    ↪Create()
        val ourPubKey: PublicKey = serviceHub.myInfo.legalIdentitiesAndCerts.first().
    ↪owningKey
        val counterpartyPubKey: PublicKey = counterparty.owningKey
        val requiredSigners: List<PublicKey> = listOf(ourPubKey, counterpartyPubKey)
        val ourCommand: Command<DummyContract.Commands.Create> = Command(commandData,
    ↪requiredSigners)
        // DOCEND 24

        // ``CommandData`` can either be:
        // 1. Of type ``TypeOnlyCommandData``, in which case it only
        //     serves to attach signers to the transaction and possibly
        //     fork the contract's verification logic.
        val typeOnlyCommandData: TypeOnlyCommandData = DummyContract.Commands.Create()
        // 2. Include additional data which can be used by the contract
        //     during verification, alongside fulfilling the roles above.
        val commandWithData: CommandData = Cash.Commands.Issue()

        // Attachments are identified by their hash.
        // The attachment with the corresponding hash must have been
        // uploaded ahead of time via the node's RPC interface.
        // DOCSTART 25
        val ourAttachment: SecureHash = SecureHash.sha256("DummyAttachment")
        // DOCEND 25

        // Time windows can have a start and end time, or be open at either end.
        // DOCSTART 26
        val ourTimeWindow: TimeWindow = TimeWindow.between(Instant.MIN, Instant.MAX)
        val ourAfter: TimeWindow = TimeWindow.fromOnly(Instant.MIN)
        val ourBefore: TimeWindow = TimeWindow.untilOnly(Instant.MAX)
        // DOCEND 26

        // We can also define a time window as an ``Instant`` +/- a time
        // tolerance (e.g. 30 seconds):
        // DOCSTART 42
        val ourTimeWindow2: TimeWindow = TimeWindow.withTolerance(serviceHub.clock.
    ↪instant(), 30.seconds)
        // DOCEND 42
        // Or as a start-time plus a duration:
        // DOCSTART 43
        val ourTimeWindow3: TimeWindow = TimeWindow.fromStartAndDuration(serviceHub.
    ↪clock.instant(), 30.seconds)
        // DOCEND 43

        /**
         * TRANSACTION BUILDING *
         */
        progressTracker.currentStep = TX_BUILDING

        // If our transaction has input states or a time-window, we must instantiate
    ↪it with a

```

(continues on next page)

(continued from previous page)

```

// notary.
// DOCSTART 19
val txBuilder: TransactionBuilder = TransactionBuilder(specificNotary)
// DOCEND 19

// Otherwise, we can choose to instantiate it without one:
// DOCSTART 46
val txBuilderNoNotary: TransactionBuilder = TransactionBuilder()
// DOCEND 46

// We add items to the transaction builder using ``TransactionBuilder``.
→withItems`:
    // DOCSTART 27
    txBuilder.withItems(
        // Inputs, as ``StateAndRef``'s that reference the outputs of previous
→transactions
        ourStateAndRef,
        // Outputs, as ``StateAndContract``'s
        ourOutput,
        // Commands, as ``Command``'s
        ourCommand,
        // Attachments, as ``SecureHash``'es
        ourAttachment,
        // A time-window, as ``TimeWindow``
        ourTimeWindow
    )
    // DOCEND 27

// We can also add items using methods for the individual components.

// The individual methods for adding input states and attachments:
// DOCSTART 28
txBuilder.addInputState(ourStateAndRef)
txBuilder.addAttachment(ourAttachment)
// DOCEND 28

// An output state can be added as a ``ContractState``, contract class name
→and notary.
// DOCSTART 49
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID,_
→specificNotary)
// DOCEND 49
// We can also leave the notary field blank, in which case the transaction's
→default
    // notary is used.
    // DOCSTART 50
    txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID)
    // DOCEND 50
    // Or we can add the output state as a ``TransactionState``, which already
→specifies
    // the output's contract and notary.
    // DOCSTART 51
val txState: TransactionState<DummyState> = TransactionState(ourOutputState,_
→DummyContract.PROGRAM_ID, specificNotary)
// DOCEND 51

// Commands can be added as ``Command``'s.

```

(continues on next page)

(continued from previous page)

```

// DOCSTART 52
txBuilder.addCommand(ourCommand)
// DOCEND 52
// Or as ``CommandData`` and a ``vararg PublicKey``.
// DOCSTART 53
txBuilder.addCommand(commandData, ourPubKey, counterpartyPubKey)
// DOCEND 53

// We can set a time-window directly.
// DOCSTART 44
txBuilder.setTimeWindow(ourTimeWindow)
// DOCEND 44
// Or as a start time plus a duration (e.g. 45 seconds).
// DOCSTART 45
txBuilder.setTimeWindow(serviceHub.clock.instant(), 45.seconds)
// DOCEND 45

/**-
 * TRANSACTION SIGNING *
-----*/
progressTracker.currentStep = TX_SIGNING

// We finalise the transaction by signing it, converting it into a
// ``SignedTransaction``.
// DOCSTART 29
val onceSignedTx: SignedTransaction = serviceHub.
→signInitialTransaction(txBuilder)
// DOCEND 29
// We can also sign the transaction using a different public key:
// DOCSTART 30
val otherIdentity: PartyAndCertificate = serviceHub.keyManagementService.
→freshKeyAndCert(ourIdentityAndCert, false)
val onceSignedTx2: SignedTransaction = serviceHub.
→signInitialTransaction(txBuilder, otherIdentity.owningKey)
// DOCEND 30

// If instead this was a ``SignedTransaction`` that we'd received
// from a counterparty and we needed to sign it, we would add our
// signature using:
// DOCSTART 38
val twiceSignedTx: SignedTransaction = serviceHub.addSignature(onceSignedTx)
// DOCEND 38
// Or, if we wanted to use a different public key:
val otherIdentity2: PartyAndCertificate = serviceHub.keyManagementService.
→freshKeyAndCert(ourIdentityAndCert, false)
// DOCSTART 39
val twiceSignedTx2: SignedTransaction = serviceHub.addSignature(onceSignedTx, ↴
→otherIdentity2.owningKey)
// DOCEND 39

// We can also generate a signature over the transaction without
// adding it to the transaction itself. We may do this when
// sending just the signature in a flow instead of returning the
// entire transaction with our signature. This way, the receiving
// node does not need to check we haven't changed anything in the
// transaction.
// DOCSTART 40

```

(continues on next page)

(continued from previous page)

```

val sig: TransactionSignature = serviceHub.createSignature(onceSignedTx)
// DOCEND 40
// And again, if we wanted to use a different public key:
// DOCSTART 41
val sig2: TransactionSignature = serviceHub.createSignature(onceSignedTx, ↵
otherIdentity2.owningKey)
// DOCEND 41

// In practice, however, the process of gathering every signature
// but the first can be automated using ``CollectSignaturesFlow``.
// See the "Gathering Signatures" section below.

/**-----*
 * TRANSACTION VERIFICATION *
-----*/
progressTracker.currentStep = TX_VERIFICATION

// Verifying a transaction will also verify every transaction in
// the transaction's dependency chain, which will require
// transaction data access on counterparty's node. The
// ``SendTransactionFlow`` can be used to automate the sending and
// data vending process. The ``SendTransactionFlow`` will listen
// for data request until the transaction is resolved and verified
// on the other side:
// DOCSTART 12
subFlow(SendTransactionFlow(counterpartySession, twiceSignedTx))

// Optional request verification to further restrict data access.
subFlow(object : SendTransactionFlow(counterpartySession, twiceSignedTx) {
    override fun verifyDataRequest(dataRequest: FetchDataFlow.Request.Data) {
        // Extra request verification.
    }
})
// DOCEND 12

// We can receive the transaction using ``ReceiveTransactionFlow``,
// which will automatically download all the dependencies and verify
// the transaction
// DOCSTART 13
val verifiedTransaction = subFlow(ReceiveTransactionFlow(counterpartySession))
// DOCEND 13

// We can also send and receive a `StateAndRef` dependency chain
// and automatically resolve its dependencies.
// DOCSTART 14
subFlow(SendStateAndRefFlow(counterpartySession, dummyStates))

// On the receive side ...
val resolvedStateAndRef = subFlow(ReceiveStateAndRefFlow<DummyState>
(counterpartySession))
// DOCEND 14

// We can now verify the transaction to ensure that it satisfies
// the contracts of all the transaction's input and output states.
// DOCSTART 33
twiceSignedTx.verify(serviceHub)
// DOCEND 33

```

(continues on next page)

(continued from previous page)

```

// We'll often want to perform our own additional verification
// too. Just because a transaction is valid based on the contract
// rules and requires our signature doesn't mean we have to
// sign it! We need to make sure the transaction represents an
// agreement we actually want to enter into.

// To do this, we need to convert our ``SignedTransaction``
// into a ``LedgerTransaction``. This will use our ServiceHub
// to resolve the transaction's inputs and attachments into
// actual objects, rather than just references.
// DOCSTART 32
val ledgerTx: LedgerTransaction = twiceSignedTx.
↳toLedgerTransaction(serviceHub)
// DOCEND 32

// We can now perform our additional verification.
// DOCSTART 34
val outputState: DummyState = ledgerTx.outputsOfType<DummyState>().single()
if (outputState.magicNumber == 777) {
    // ``FlowException`` is a special exception type. It will be
    // propagated back to any counterparty flows waiting for a
    // message from this flow, notifying them that the flow has
    // failed.
    throw FlowException("We expected a magic number of 777.")
}
// DOCEND 34

// Of course, if you are not a required signer on the transaction,
// you have no power to decide whether it is valid or not. If it
// requires signatures from all the required signers and is
// contractually valid, it's a valid ledger update.

/*
 * GATHERING SIGNATURES *
*/
progressTracker.currentStep = SIGS_GATHERING

// The list of parties who need to sign a transaction is dictated
// by the transaction's commands. Once we've signed a transaction
// ourselves, we can automatically gather the signatures of the
// other required signers using ``CollectSignaturesFlow``.
// The responder flow will need to call ``SignTransactionFlow``.
// DOCSTART 15
val fullySignedTx: SignedTransaction =
↳subFlow(CollectSignaturesFlow(twiceSignedTx, setOf(counterpartySession, ↳
↳regulatorSession), SIGS_GATHERING.childProgressTracker()))
// DOCEND 15

/*
 * VERIFYING SIGNATURES *
*/
progressTracker.currentStep = VERIFYING_SIGS

// We can verify that a transaction has all the required
// signatures, and that they're all valid, by running:
// DOCSTART 35

```

(continues on next page)

(continued from previous page)

```

fullySignedTx.verifyRequiredSignatures()
// DOCEND 35

// If the transaction is only partially signed, we have to pass in
// a vararg of the public keys corresponding to the missing
// signatures, explicitly telling the system not to check them.
// DOCSTART 36
onceSignedTx.verifySignaturesExcept(counterpartyPubKey)
// DOCEND 36

// There is also an overload of ``verifySignaturesExcept`` which accepts
// a `Collection` of the public keys corresponding to the missing
// signatures.
// DOCSTART 54
onceSignedTx.verifySignaturesExcept(listOf(counterpartyPubKey))
// DOCEND 54

// We can also choose to only check the signatures that are
// present. BE VERY CAREFUL - this function provides no guarantees
// that the signatures are correct, or that none are missing.
// DOCSTART 37
twiceSignedTx.checkSignaturesAreValid()
// DOCEND 37

/*-----
 * FINALISING THE TRANSACTION *
-----*/
progressTracker.currentStep = FINALISATION

// We notarise the transaction and get it recorded in the vault of
// the participants of all the transaction's states.
// DOCSTART 09
val notarisedTx1: SignedTransaction = subFlow(FinalityFlow(fullySignedTx,
    listOf(counterpartySession), FINALISATION.childProgressTracker()))
// DOCEND 09
// We can also choose to send it to additional parties who aren't one
// of the state's participants.
// DOCSTART 10
val partySessions: List<FlowSession> = listOf(counterpartySession,
    initiateFlow(regulator))
val notarisedTx2: SignedTransaction = subFlow(FinalityFlow(fullySignedTx,
    partySessions, FINALISATION.childProgressTracker()))
// DOCEND 10

// DOCSTART FlowSession porting
send(regulator, Any()) // Old API
// becomes
val session = initiateFlow(regulator)
session.send(Any())
// DOCEND FlowSession porting
}

}

// ``ResponderFlow`` is our second flow, and will communicate with
// ``InitiatorFlow``.
// We mark ``ResponderFlow`` as an ``InitiatedByFlow``, meaning that it
// can only be started in response to a message from its initiating flow.

```

(continues on next page)

(continued from previous page)

```
// That's ``InitiatorFlow`` in this case.
// Each node also has several flow pairs registered by default - see
// ``AbstractNode.installCoreFlows``.
@InitiatedBy(InitiatorFlow::class)
class ResponderFlow(val counterpartySession: FlowSession) : FlowLogic<Unit>() {

    companion object {
        object RECEIVING_AND_SENDING_DATA : Step("Sending data between parties.")
        object SIGNING : Step("Responding to CollectSignaturesFlow.")
        object FINALISATION : Step("Finalising a transaction.")

        fun tracker() = ProgressTracker(
            RECEIVING_AND_SENDING_DATA,
            SIGNING,
            FINALISATION
        )
    }

    override val progressTracker: ProgressTracker = tracker()

    @Suspendable
    override fun call() {
        // The ``ResponderFlow`` has all the same APIs available. It looks
        // up network information, sends and receives data, and constructs
        // transactions in exactly the same way.

        /**
         * SENDING AND RECEIVING DATA *
         */
        progressTracker.currentStep = RECEIVING_AND_SENDING_DATA

        // We need to respond to the messages sent by the initiator:
        // 1. They sent us an ``Any`` instance
        // 2. They waited to receive an ``Integer`` instance back
        // 3. They sent a ``String`` instance and waited to receive a
        //    ``Boolean`` instance back
        // Our side of the flow must mirror these calls.
        // DOCSTART 08
        val any: Any = counterpartySession.receive<Any>().unwrap { data -> data }
        val string: String = counterpartySession.sendAndReceive<String>(99).unwrap { _ -> data }
        counterpartySession.send(true)
        // DOCEND 08

        /**
         * RESPONDING TO COLLECT_SIGNATURES_FLOW *
         */
        progressTracker.currentStep = SIGNING

        // The responder will often need to respond to a call to
        // ``CollectSignaturesFlow``. It does so by invoking its own
        // ``SignTransactionFlow`` subclass.
        // DOCSTART 16
        val signTransactionFlow: SignTransactionFlow = object : SignTransactionFlow(counterpartySession) {
            override fun checkTransaction(stx: SignedTransaction) = requireThat {
                // Any additional checking we see fit...
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        val outputState = stx.tx.outputsOfType<DummyState>().single()
        require(outputState.magicNumber == 777)
    }
}

val idOfTxWeSigned = subFlow(signTransactionFlow).id
// DOCEND 16

/*
 * FINALISING THE TRANSACTION *
-----*/
progressTracker.currentStep = FINALISATION

// As the final step the responder waits to receive the notarised transaction
// from the sending party
// Since it knows the ID of the transaction it just signed, the transaction
// ID is specified to ensure the correct
// transaction is received and recorded.
// DOCSTART ReceiveFinalityFlow
subFlow(ReceiveFinalityFlow(counterpartySession, expectedTxId =
idOfTxWeSigned))
// DOCEND ReceiveFinalityFlow
}
}

```

```

package net.corda.docs.java;

import co.paralleluniverse.fibers.Suspendable;
import com.google.common.collect.ImmutableList;
import net.corda.core.contracts.*;
import net.corda.core.crypto.SecureHash;
import net.corda.core.crypto.TransactionSignature;
import net.corda.core.flows.*;
import net.corda.core.identity.CordaX500Name;
import net.corda.core.identity.Party;
import net.corda.core.identity.PartyAndCertificate;
import net.corda.core.internal.FetchDataFlow;
import net.corda.core.node.services.Vault;
import net.corda.core.node.services.Vault.Page;
import net.corda.core.node.services.vault.QueryCriteria.VaultQueryCriteria;
import net.corda.core.transactions.LedgerTransaction;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;
import net.corda.core.utilities.ProgressTracker.Step;
import net.corda.core.utilities.UntrustworthyData;
import net.corda.finance.contracts.asset.Cash;
import net.corda.testing.contracts.DummyContract;
import net.corda.testing.contracts.DummyState;
import org.jetbrains.annotations.NotNull;

import java.security.GeneralSecurityException;
import java.security.PublicKey;
import java.time.Duration;
import java.time.Instant;
import java.util.Arrays;

```

(continues on next page)

(continued from previous page)

```

import java.util.List;

import static com.google.common.base.Preconditions.checkNotNull;
import static java.util.Collections.*;
import static net.corda.core.contracts.ContractsDSL.requireThat;
import static net.corda.core.crypto.Crypto.generateKeyPair;

@SuppressWarnings("unused")
public class FlowCookbook {
    // ``InitiatorFlow`` is our first flow, and will communicate with
    // ``ResponderFlow``, below.
    // We mark ``InitiatorFlow`` as an ``InitiatingFlow``, allowing it to be
    // started directly by the node.
    @InitiatingFlow
    // We also mark ``InitiatorFlow`` as ``StartableByRPC``, allowing the
    // node's owner to start the flow via RPC.
    @StartableByRPC
    // Every flow must subclass ``FlowLogic``. The generic indicates the
    // flow's return type.
    public static class InitiatorFlow extends FlowLogic<Void> {

        private final boolean arg1;
        private final int arg2;
        private final Party counterparty;
        private final Party regulator;

        public InitiatorFlow(boolean arg1, int arg2, Party counterparty, Party regulator) {
            this.arg1 = arg1;
            this.arg2 = arg2;
            this.counterparty = counterparty;
            this.regulator = regulator;
        }

        /**
         * WIRING UP THE PROGRESS TRACKER *
         */
        // Giving our flow a progress tracker allows us to see the flow's
        // progress visually in our node's CRaSH shell.
        // DOCSTART 17
        private static final Step ID_OTHER_NODES = new Step("Identifying other nodes",
        on the network.");
        private static final Step SENDING_AND RECEIVING_DATA = new Step("Sending data",
        between parties.");
        private static final Step EXTRACTING_VAULT_STATES = new Step("Extracting,
        states from the vault.");
        private static final Step OTHER_TX_COMPONENTS = new Step("Gathering a,
        transaction's other components.");
        private static final Step TX_BUILDING = new Step("Building a transaction.");
        private static final Step TX_SIGNING = new Step("Signing a transaction.");
        private static final Step TX_VERIFICATION = new Step("Verifying a transaction.
        ");
        private static final Step SIGS_GATHERING = new Step("Gathering a transaction
        's signatures.");
        // Wiring up a child progress tracker allows us to see the
        // subflow's progress steps in our flow's progress tracker.
        @Override
    }
}

```

(continues on next page)

(continued from previous page)

```

        public ProgressTracker childProgressTracker() {
            return CollectSignaturesFlow.tracker();
        }
    };
    private static final Step VERIFYING_SIGS = new Step("Verifying a transaction
→'s signatures.");
    private static final Step FINALISATION = new Step("Finalising a transaction.
→");
}

@Override
public ProgressTracker childProgressTracker() {
    return FinalityFlow.tracker();
}
};

private final ProgressTracker progressTracker = new ProgressTracker(
    ID_OTHER_NODES,
    SENDING_AND RECEIVING_DATA,
    EXTRACTING_VAULT_STATES,
    OTHER_TX_COMPONENTS,
    TX_BUILDING,
    TX_SIGNING,
    TX_VERIFICATION,
    SIGS_GATHERING,
    FINALISATION
);
// DOCEND 17

@suspendable
@Override
public Void call() throws FlowException {
    // We'll be using a dummy public key for demonstration purposes.
    PublicKey dummyPubKey = generateKeyPair().getPublic();

    /**
     * IDENTIFYING OTHER NODES *
     */
    // DOCSTART 18
    progressTracker.setCurrentStep(ID_OTHER_NODES);
    // DOCEND 18

    // A transaction generally needs a notary:
    // - To prevent double-spends if the transaction has inputs
    // - To serve as a timestamping authority if the transaction has a
    //   time-window
    // We retrieve a notary from the network map.
    // DOCSTART 01
    CordaX500Name notaryName = new CordaX500Name("Notary Service", "London",
    "GB");
    Party specificNotary = getServiceHub().getNetworkMapCache().
    ↪getNotary(notaryName);
    // Alternatively, we can pick an arbitrary notary from the notary
    // list. However, it is always preferable to specify the notary
    // explicitly, as the notary list might change when new notaries are
    // introduced, or old ones decommissioned.
    Party firstNotary = getServiceHub().getNetworkMapCache().
    ↪getNotaryIdentities().get(0);
    // DOCEND 01
}

```

(continues on next page)

(continued from previous page)

```

// We may also need to identify a specific counterparty. We do so
// using the identity service.
// DOCSTART 02
CordaX500Name counterPartyName = new CordaX500Name("NodeA", "London", "GB
→");
Party namedCounterparty = getServiceHub().getIdentityService() .
←wellKnownPartyFromX500Name(counterPartyName);
Party keyedCounterparty = getServiceHub().getIdentityService() .
←partyFromKey(dummyPubKey);
// DOCEND 02

/*
-----*
 * SENDING AND RECEIVING DATA *
-----*/
progressTracker.setCurrentStep(SENDING_AND_RECEIVING_DATA);

// We start by initiating a flow session with the counterparty. We
// will use this session to send and receive messages from the
// counterparty.
// DOCSTART initiateFlow
FlowSession counterpartySession = initiateFlow(counterparty);
// DOCEND initiateFlow

// We can send arbitrary data to a counterparty.
// If this is the first ``send``, the counterparty will either:
// 1. Ignore the message if they are not registered to respond
// to messages from this flow.
// 2. Start the flow they have registered to respond to this flow,
// and run the flow until the first call to ``receive``, at
// which point they process the message.
// In other words, we are assuming that the counterparty is
// registered to respond to this flow, and has a corresponding
// ``receive`` call.
// DOCSTART 04
counterpartySession.send(new Object());
// DOCEND 04

// We can wait to receive arbitrary data of a specific type from a
// counterparty. Again, this implies a corresponding ``send`` call
// in the counterparty's flow. A few scenarios:
// - We never receive a message back. In the current design, the
// flow is paused until the node's owner kills the flow.
// - Instead of sending a message back, the counterparty throws a
// ``FlowException``. This exception is propagated back to us,
// and we can use the error message to establish what happened.
// - We receive a message back, but it's of the wrong type. In
// this case, a ``FlowException`` is thrown.
// - We receive back a message of the correct type. All is good.
//
// Upon calling ``receive()`` (or ``sendAndReceive()``), the
// ``FlowLogic`` is suspended until it receives a response.
//
// We receive the data wrapped in an ``UntrustworthyData``
// instance. This is a reminder that the data we receive may not
// be what it appears to be! We must unwrap the
// ``UntrustworthyData`` using a lambda.

```

(continues on next page)

(continued from previous page)

```

// DOCSTART 05
UntrustworthyData<Integer> packet1 = counterpartySession.receive(Integer.
˓→class);
Integer integer = packet1.unwrap(data -> {
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    return data;
});
// DOCEND 05

// We can also use a single call to send data to a counterparty
// and wait to receive data of a specific type back. The type of
// data sent doesn't need to match the type of the data received
// back.
// DOCSTART 07
UntrustworthyData<Boolean> packet2 = counterpartySession.
˓→sendAndReceive(Boolean.class, "You can send and receive any class!");
Boolean bool = packet2.unwrap(data -> {
    // Perform checking on the object received.
    // T O D O: Check the received object.
    // Return the object.
    return data;
});
// DOCEND 07

// We're not limited to sending to and receiving from a single
// counterparty. A flow can send messages to as many parties as it
// likes, and each party can invoke a different response flow.
// DOCSTART 06
FlowSession regulatorSession = initiateFlow(regulator);
regulatorSession.send(new Object());
UntrustworthyData<Object> packet3 = regulatorSession.receive(Object.
˓→class);
// DOCEND 06

/*****
 * EXTRACTING STATES FROM THE VAULT *
 ****/
progressTracker.setCurrentStep(EXTRACTING_VAULT_STATES);

// Let's assume there are already some ``DummyState``'s in our
// node's vault, stored there as a result of running past flows,
// and we want to consume them in a transaction. There are many
// ways to extract these states from our vault.

// For example, we would extract any unconsumed ``DummyState``'s
// from our vault as follows:
VaultQueryCriteria criteria = new VaultQueryCriteria(Vault.StateStatus.
˓→UNCONSUMED);
Page<DummyState> results = getServiceHub().getVaultService().
˓→queryBy(DummyState.class, criteria);
List<StateAndRef<DummyState>> dummyStates = results.getStates();

// For a full list of the available ways of extracting states from
// the vault, see the Vault Query docs page.

```

(continues on next page)

(continued from previous page)

```

// When building a transaction, input states are passed in as
// ``StateRef`` instances, which pair the hash of the transaction
// that generated the state with the state's index in the outputs
// of that transaction. In practice, we'd pass the transaction hash
// or the ``StateRef`` as a parameter to the flow, or extract the
// ``StateRef`` from our vault.
// DOCSTART 20
StateRef ourStateRef = new StateRef(SecureHash.sha256(
    "DummyTransactionHash"), 0);
// DOCEND 20
// A ``StateAndRef`` pairs a ``StateRef`` with the state it points to.
// DOCSTART 21
StateAndRef ourStateAndRef = getServiceHub().toStateAndRef(ourStateRef);
// DOCEND 21

/*-----*
 * GATHERING OTHER TRANSACTION COMPONENTS *
-----*/
progressTracker.setCurrentStep(OTHER_TX_COMPONENTS);

// Reference input states are constructed from StateAndRefs.
// DOCSTART 55
ReferencedStateAndRef referenceState = ourStateAndRef.referenced();
// DOCEND 55
// Output states are constructed from scratch.
// DOCSTART 22
DummyState ourOutputState = new DummyState();
// DOCEND 22
// Or as copies of other states with some properties changed.
// DOCSTART 23
DummyState ourOtherOutputState = ourOutputState.copy(77);
// DOCEND 23

// We then need to pair our output state with a contract.
// DOCSTART 47
StateAndContract ourOutput = new StateAndContract(ourOutputState,
    DummyContract.PROGRAM_ID);
// DOCEND 47

// Commands pair a ``CommandData`` instance with a list of
// public keys. To be valid, the transaction requires a signature
// matching every public key in all of the transaction's commands.
// DOCSTART 24
DummyContract.Commands.Create commandData = new DummyContract.Commands.
    Create();
    PublicKey ourPubKey = getServiceHub().getMyInfo().
    getLegalIdentitiesAndCerts().get(0).getOwningKey();
    PublicKey counterpartyPubKey = counterparty.getOwningKey();
    List<PublicKey> requiredSigners = ImmutableList.of(ourPubKey,
        counterpartyPubKey);
    Command<DummyContract.Commands.Create> ourCommand = new Command<>(
        commandData, requiredSigners);
// DOCEND 24

// ``CommandData`` can either be:
// 1. Of type ``TypeOnlyCommandData``, in which case it only
//     serves to attach signers to the transaction and possibly

```

(continues on next page)

(continued from previous page)

```

    // fork the contract's verification logic.
    TypeOnlyCommandData typeOnlyCommandData = new DummyContract.Commands.
    ↪Create();
    // 2. Include additional data which can be used by the contract
    // during verification, alongside fulfilling the roles above
    CommandData commandWithData = new Cash.Commands.Issue();

    // Attachments are identified by their hash.
    // The attachment with the corresponding hash must have been
    // uploaded ahead of time via the node's RPC interface.
    // DOCSTART 25
    SecureHash ourAttachment = SecureHash.sha256("DummyAttachment");
    // DOCEND 25

    // Time windows represent the period of time during which a
    // transaction must be notarised. They can have a start and an end
    // time, or be open at either end.
    // DOCSTART 26
    TimeWindow ourTimeWindow = TimeWindow.between(Instant.MIN, Instant.MAX);
    TimeWindow ourAfter = TimeWindow.fromOnly(Instant.MIN);
    TimeWindow ourBefore = TimeWindow.untilOnly(Instant.MAX);
    // DOCEND 26

    // We can also define a time window as an ``Instant`` +/- a time
    // tolerance (e.g. 30 seconds):
    // DOCSTART 42
    TimeWindow ourTimeWindow2 = TimeWindow.withTolerance(getServiceHub().
    ↪getClock().instant(), Duration.ofSeconds(30));
    // DOCEND 42
    // Or as a start-time plus a duration:
    // DOCSTART 43
    TimeWindow ourTimeWindow3 = TimeWindow.
    ↪fromStartAndDuration(getServiceHub().getClock().instant(), Duration.ofSeconds(30));
    // DOCEND 43

    /*****
     * TRANSACTION BUILDING *
     *****/
    progressTracker.setCurrentStep(TX_BUILDING);

    // If our transaction has input states or a time-window, we must
    ↪instantiate it with a
    // notary.
    // DOCSTART 19
    TransactionBuilder txBuilder = new TransactionBuilder(specificNotary);
    // DOCEND 19

    // Otherwise, we can choose to instantiate it without one:
    // DOCSTART 46
    TransactionBuilder txBuilderNoNotary = new TransactionBuilder();
    // DOCEND 46

    // We add items to the transaction builder using ``TransactionBuilder.
    ↪withItems``:
    // DOCSTART 27
    txBuilder.withItems(
        // Inputs, as ``StateAndRef``'s that reference to the outputs of
        ↪previous transactions

```

(continues on next page)

(continued from previous page)

```

        ourStateAndRef,
        // Outputs, as ``StateAndContract``s
        ourOutput,
        // Commands, as ``Command``s
        ourCommand,
        // Attachments, as ``SecureHash``es
        ourAttachment,
        // A time-window, as ``TimeWindow``
        ourTimeWindow
    );
// DOCEND 27

    // We can also add items using methods for the individual components.

    // The individual methods for adding input states and attachments:
// DOCSTART 28
txBuilder.addInputState(ourStateAndRef);
txBuilder.addAttachment(ourAttachment);
// DOCEND 28

    // An output state can be added as a ``ContractState``, contract class
    ↪name and notary.
    // DOCSTART 49
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID,
    ↪specificNotary);
    // DOCEND 49
    // We can also leave the notary field blank, in which case the transaction
    ↪'s default
        // notary is used.
    // DOCSTART 50
txBuilder.addOutputState(ourOutputState, DummyContract.PROGRAM_ID);
    // DOCEND 50
    // Or we can add the output state as a ``TransactionState``, which
    ↪already specifies
        // the output's contract and notary.
    // DOCSTART 51
TransactionState txState = new TransactionState(ourOutputState,
    ↪DummyContract.PROGRAM_ID, specificNotary);
    // DOCEND 51

    // Commands can be added as ``Command``s.
    // DOCSTART 52
txBuilder.addCommand(ourCommand);
    // DOCEND 52
    // Or as ``CommandData`` and a ``vararg PublicKey``.
    // DOCSTART 53
txBuilder.addCommand(commandData, ourPubKey, counterpartyPubKey);
    // DOCEND 53

    // We can set a time-window directly.
    // DOCSTART 44
txBuilder.setTimeWindow(ourTimeWindow);
    // DOCEND 44
    // Or as a start time plus a duration (e.g. 45 seconds).
    // DOCSTART 45
txBuilder.setTimeWindow(getServiceHub().getClock().instant(), Duration.
    ↪ofSeconds(45));

```

(continues on next page)

(continued from previous page)

```

// DOCEND 45

/*
 * TRANSACTION SIGNING *
*/
progressTracker.setCurrentStep(TX_SIGNING);

// We finalise the transaction by signing it,
// converting it into a ``SignedTransaction``.
// DOCSTART 29
SignedTransaction onceSignedTx = getServiceHub() .
→signInitialTransaction(txBuilder);
// DOCEND 29
// We can also sign the transaction using a different public key:
// DOCSTART 30
PartyAndCertificate otherIdentity = getServiceHub() .
→getKeyManagementService().freshKeyAndCert(getOurIdentityAndCert(), false);
SignedTransaction onceSignedTx2 = getServiceHub() .
→signInitialTransaction(txBuilder, otherIdentity.getOwningKey());
// DOCEND 30

// If instead this was a ``SignedTransaction`` that we'd received
// from a counterparty and we needed to sign it, we would add our
// signature using:
// DOCSTART 38
SignedTransaction twiceSignedTx = getServiceHub() .
→addSignature(onceSignedTx);
// DOCEND 38
// Or, if we wanted to use a different public key:
PartyAndCertificate otherIdentity2 = getServiceHub() .
→getKeyManagementService().freshKeyAndCert(getOurIdentityAndCert(), false);
// DOCSTART 39
SignedTransaction twiceSignedTx2 = getServiceHub() .
→addSignature(onceSignedTx, otherIdentity2.getOwningKey());
// DOCEND 39

// We can also generate a signature over the transaction without
// adding it to the transaction itself. We may do this when
// sending just the signature in a flow instead of returning the
// entire transaction with our signature. This way, the receiving
// node does not need to check we haven't changed anything in the
// transaction.
// DOCSTART 40
TransactionSignature sig = getServiceHub().createSignature(onceSignedTx);
// DOCEND 40
// And again, if we wanted to use a different public key:
// DOCSTART 41
TransactionSignature sig2 = getServiceHub().createSignature(onceSignedTx, ↴
otherIdentity2.getOwningKey());
// DOCEND 41

/*
 * TRANSACTION VERIFICATION *
*/
progressTracker.setCurrentStep(TX_VERIFICATION);

// Verifying a transaction will also verify every transaction in

```

(continues on next page)

(continued from previous page)

```

// the transaction's dependency chain, which will require
// transaction data access on counterparty's node. The
// ``SendTransactionFlow`` can be used to automate the sending and
// data vending process. The ``SendTransactionFlow`` will listen
// for data request until the transaction is resolved and verified
// on the other side:
// DOCSTART 12
subFlow(new SendTransactionFlow(counterpartySession, twiceSignedTx));

// Optional request verification to further restrict data access.
subFlow(new SendTransactionFlow(counterpartySession, twiceSignedTx) {
    @Override
    protected void verifyDataRequest(@NotNull FetchDataFlow.Request.Data_
→dataRequest) {
        // Extra request verification.
    }
});
// DOCEND 12

// We can receive the transaction using ``ReceiveTransactionFlow``,
// which will automatically download all the dependencies and verify
// the transaction and then record in our vault
// DOCSTART 13
SignedTransaction verifiedTransaction = subFlow(new_
→ReceiveTransactionFlow(counterpartySession));
// DOCEND 13

// We can also send and receive a `StateAndRef` dependency chain and_
→automatically resolve its dependencies.
// DOCSTART 14
subFlow(new SendStateAndRefFlow(counterpartySession, dummyStates));

// On the receive side ...
List<StateAndRef<DummyState>> resolvedStateAndRef = subFlow(new_
→ReceiveStateAndRefFlow<>(counterpartySession));
// DOCEND 14

try {

    // We can now verify the transaction to ensure that it satisfies
    // the contracts of all the transaction's input and output states.
    // DOCSTART 33
    twiceSignedTx.verify(getServiceHub());
    // DOCEND 33

    // We'll often want to perform our own additional verification
    // too. Just because a transaction is valid based on the contract
    // rules and requires our signature doesn't mean we have to
    // sign it! We need to make sure the transaction represents an
    // agreement we actually want to enter into.

    // To do this, we need to convert our ``SignedTransaction``
    // into a ``LedgerTransaction``. This will use our ServiceHub
    // to resolve the transaction's inputs and attachments into
    // actual objects, rather than just references.
    // DOCSTART 32
    LedgerTransaction ledgerTx = twiceSignedTx.
→toLedgerTransaction(getServiceHub());
}

```

(continues on next page)

(continued from previous page)

```

// DOCEND 32

// We can now perform our additional verification.
// DOCSTART 34
DummyState outputState = ledgerTx.outputsOfType(DummyState.class).
↳get(0);
if (outputState.getMagicNumber() != 777) {
    // ``FlowException`` is a special exception type. It will be
    // propagated back to any counterparty flows waiting for a
    // message from this flow, notifying them that the flow has
    // failed.
    throw new FlowException("We expected a magic number of 777.");
}
// DOCEND 34

} catch (GeneralSecurityException e) {
    // Handle this as required.
}

// Of course, if you are not a required signer on the transaction,
// you have no power to decide whether it is valid or not. If it
// requires signatures from all the required signers and is
// contractually valid, it's a valid ledger update.

/*-----*
 * GATHERING SIGNATURES *
-----*/
progressTracker.setCurrentStep(SIGS_GATHERING);

// The list of parties who need to sign a transaction is dictated
// by the transaction's commands. Once we've signed a transaction
// ourselves, we can automatically gather the signatures of the
// other required signers using ``CollectSignaturesFlow``.
// The responder flow will need to call ``SignTransactionFlow``.
// DOCSTART 15
SignedTransaction fullySignedTx = subFlow(new_
↳CollectSignaturesFlow(twiceSignedTx, emptySet(), SIGS_GATHERING,
↳childProgressTracker()));
// DOCEND 15

/*-----*
 * VERIFYING SIGNATURES *
-----*/
progressTracker.setCurrentStep(VERIFYING_SIGS);

try {

    // We can verify that a transaction has all the required
    // signatures, and that they're all valid, by running:
    // DOCSTART 35
    fullySignedTx.verifyRequiredSignatures();
    // DOCEND 35

    // If the transaction is only partially signed, we have to pass in
    // a vararg of the public keys corresponding to the missing
    // signatures, explicitly telling the system not to check them.
    // DOCSTART 36

```

(continues on next page)

(continued from previous page)

```

onceSignedTx.verifySignaturesExcept(counterpartyPubKey);
// DOCEND 36

// There is also an overload of ``verifySignaturesExcept`` which
→accepts
    // a ``Collection`` of the public keys corresponding to the missing
    // signatures. In the example below, we could also use
    // ``Arrays.asList(counterpartyPubKey)`` instead of
    // ``Collections.singletonList(counterpartyPubKey)``.
    // DOCSTART 54
    onceSignedTx.
→verifySignaturesExcept(singletonList(counterpartyPubKey));
    // DOCEND 54

    // We can also choose to only check the signatures that are
    // present. BE VERY CAREFUL - this function provides no guarantees
    // that the signatures are correct, or that none are missing.
    // DOCSTART 37
    twiceSignedTx.checkSignaturesAreValid();
    // DOCEND 37
} catch (GeneralSecurityException e) {
    // Handle this as required.
}

/*
 * FINALISING THE TRANSACTION *
*/
progressTracker.setCurrentStep(FINALISATION);

// We notarise the transaction and get it recorded in the vault of
// the participants of all the transaction's states.
// DOCSTART 09
SignedTransaction notarisedTx1 = subFlow(new FinalityFlow(fullySignedTx,
→singleton(counterpartySession), FINALISATION.childProgressTracker()));
// DOCEND 09
// We can also choose to send it to additional parties who aren't one
// of the state's participants.
// DOCSTART 10
List<FlowSession> partySessions = Arrays.asList(counterpartySession,
→initiateFlow(regulator));
SignedTransaction notarisedTx2 = subFlow(new FinalityFlow(fullySignedTx,
→partySessions, FINALISATION.childProgressTracker()));
// DOCEND 10

// DOCSTART FlowSession porting
send(regulator, new Object()); // Old API
// becomes
FlowSession session = initiateFlow(regulator);
session.send(new Object());
// DOCEND FlowSession porting

return null;
}
}

// ``ResponderFlow`` is our second flow, and will communicate with
// ``InitiatorFlow``.

```

(continues on next page)

(continued from previous page)

```

// We mark ``ResponderFlow`` as an ``InitiatedByFlow``, meaning that it
// can only be started in response to a message from its initiating flow.
// That's ``InitiatorFlow`` in this case.
// Each node also has several flow pairs registered by default - see
// ``AbstractNode.installCoreFlows``.
@InitiatedBy(InitiatorFlow.class)
public static class ResponderFlow extends FlowLogic<Void> {

    private final FlowSession counterpartySession;

    public ResponderFlow(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    private static final Step RECEIVING_AND_SENDING_DATA = new Step("Sending data\u2192 between parties.");
    private static final Step SIGNING = new Step("Responding to\u2192 CollectSignaturesFlow.");
    private static final Step FINALISATION = new Step("Finalising a transaction.\u2192");

    private final ProgressTracker progressTracker = new ProgressTracker(
        RECEIVING_AND_SENDING_DATA,
        SIGNING,
        FINALISATION
    );

    @Suspendable
    @Override
    public Void call() throws FlowException {
        // The ``ResponderFlow`` has all the same APIs available. It looks
        // up network information, sends and receives data, and constructs
        // transactions in exactly the same way.

        /*****
         * SENDING AND RECEIVING DATA *
         *****/
        progressTracker.setCurrentStep(RECEIVING_AND_SENDING_DATA);

        // We need to respond to the messages sent by the initiator:
        // 1. They sent us an ``Object`` instance
        // 2. They waited to receive an ``Integer`` instance back
        // 3. They sent a ``String`` instance and waited to receive a
        //    ``Boolean`` instance back
        // Our side of the flow must mirror these calls.
        // DOCSTART 08
        Object obj = counterpartySession.receive(Object.class).unwrap(data ->
    data);
        String string = counterpartySession.sendAndReceive(String.class, 99).
    unwrap(data -> data);
        counterpartySession.send(true);
        // DOCEND 08

        /*****
         * RESPONDING TO COLLECT_SIGNATURES_FLOW *
         *****/
        progressTracker.setCurrentStep(SIGNING);
    }
}

```

(continues on next page)

(continued from previous page)

```

// The responder will often need to respond to a call to
// ``CollectSignaturesFlow``. It does so by invoking its own
// ``SignTransactionFlow`` subclass.
// DOCSTART 16
class SignTxFlow extends SignTransactionFlow {
    private SignTxFlow(FlowSession otherSession, ProgressTracker_
→progressTracker) {
        super(otherSession, progressTracker);
    }

    @Override
    protected void checkTransaction(SignedTransaction stx) {
        requireThat(require -> {
            // Any additional checking we see fit...
            DummyState outputState = (DummyState) stx.getTx()._
→getOutputs().get(0).getData();
            checkArgument(outputState.getMagicNumber() == 777);
            return null;
        });
    }
}

SecureHash idOfTxWeSigned = subFlow(new SignTxFlow(counterpartySession,_
→SignTransactionFlow.tracker())).getId();
// DOCEND 16

/*
 * FINALISING THE TRANSACTION
 */
progressTracker.setCurrentStep(FINALISATION);

// As the final step the responder waits to receive the notarised_
→transaction from the sending party
// Since it knows the ID of the transaction it just signed, the_
→transaction ID is specified to ensure the correct
// transaction is received and recorded.
// DOCSTART ReceiveFinalityFlow
subFlow(new ReceiveFinalityFlow(counterpartySession, idOfTxWeSigned));
// DOCEND ReceiveFinalityFlow

return null;
}
}
}

```

**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.

## 9.1 Hello, World!

### 9.1.1 The CorDapp Template

When writing a new CorDapp, you'll generally want to start from one of the standard templates:

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

The Cordapp templates provide the boilerplate for developing a new CorDapp. CorDapps can be written in either Java or Kotlin. We will be providing the code in both languages throughout this tutorial.

Note that there's no need to download and install Corda itself. The required libraries are automatically downloaded from an online Maven repository and cached locally.

#### Downloading 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
```

```
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

For this tutorial, we will only be modifying the following files:

```
// 1. The state
contracts/src/main/java/com/template/states/TemplateState.java

// 2. The flow
workflows/src/main/java/com/template/flows/Initiator.java
```

```
// 1. The state
contracts/src/main/kotlin/com/template/states/TemplateState.kt

// 2. The flow
workflows/src/main/kotlin/com/template/flows/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`.

### 9.1.2 Writing the state

In Corda, shared facts on the blockchain 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:

```
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 blockchain.

---

**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 blockchain? 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`:

```
// Add this import:  
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)  
}
```

```
// Add this import:  
import net.corda.core.identity.Party;  
  
// Replace TemplateState's definition with:  
public class IOUState implements ContractState {  
    private final int value;  
    private final Party lender;  
    private final Party borrower;  
  
    public IOUState(int value, Party lender, Party borrower) {  
        this.value = value;  
        this.lender = lender;  
        this.borrower = borrower;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    public Party getLender() {  
        return lender;  
    }  
  
    public Party getBorrower() {  
        return borrower;  
    }  
  
    @Override
```

(continues on next page)

(continued from previous page)

```
public List<AbstractParty> getParticipants() {
    return Arrays.asList(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 blockchain, 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.

### 9.1.3 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 and sending it to the IOU's borrower so that they can record it too

We also need the borrower to receive the transaction and record it for itself. 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.

**Warning:** The execution of a flow is distributed in space and time, as the flow crosses node boundaries and each participant may have to wait for other participants to respond before it can complete its part of the overall work. While a node is waiting, the state of its flow may be persistently recorded to disk as a restorable checkpoint, enabling it to carry on where it left off when a counterparty responds. However, before a node can be upgraded to a newer version of Corda, or of your Cordapp, all flows must have completed, as there is no mechanism to upgrade a persisted flow checkpoint. It is therefore undesirable to model a long-running business process as a single flow: it should rather be broken up into a series of transactions, with flows used only to orchestrate the completion of each transaction.

## 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*.

## FlowLogic

All flows must subclass `FlowLogic`. You then define the steps taken by the flow by overriding `FlowLogic.call`.

Let's define our `IOUFlow`. Replace the definition of `Initiator` with the following:

```
// Add these imports:
import net.corda.core.contracts.Command
import net.corda.core.identity.Party
import net.corda.core.transactions.TransactionBuilder

// Replace Initiator'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.
    }
}
```

(continues on next page)

(continued from previous page)

```

val notary = serviceHub.networkMapCache.notaryIdentities[0]

// We create the transaction components.
val outputState = IOUState(iouValue, ourIdentity, otherParty)
val command = 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(command)

// We sign the transaction.
val signedTx = serviceHub.signInitialTransaction(txBuilder)

// Creating a session with the other party.
val otherPartySession = initiateFlow(otherParty)

// We finalise the transaction and then send it to the counterparty.
subFlow(FinalityFlow(signedTx, otherPartySession))
}
}

```

```

// Add these imports:
import net.corda.core.contracts.Command;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;

// Replace Initiator's definition with:
@InitiatingFlow
@StartableByRPC
public class IOUFlow extends FlowLogic<Void> {
    private final Integer iouValue;
    private final Party otherParty;

    /**
     * The progress tracker provides checkpoints indicating the progress of the flow
     →to observers.
     */
    private final ProgressTracker progressTracker = new ProgressTracker();

    public IOUFlow(Integer iouValue, Party otherParty) {
        this.iouValue = iouValue;
        this.otherParty = otherParty;
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    /**
     * The flow logic is encapsulated within the call() method.
     */
    @Suspendable

```

(continues on next page)

(continued from previous page)

```

@Override
public Void call() throws FlowException {
    // We retrieve the notary identity from the network map.
    Party notary = getServiceHub().getNetworkMapCache().getNotaryIdentities().
    ↪get(0);

    // We create the transaction components.
    IOUState outputState = new IOUState(iouValue, getOurIdentity(), otherParty);
    Command command = new Command<>(new TemplateContract.Commands.Action(), ↪
    ↪getOurIdentity().getOwningKey());

    // We create a transaction builder and add the components.
    TransactionBuilder txBuilder = new TransactionBuilder(notary)
        .addOutputState(outputState, TemplateContract.ID)
        .addCommand(command);

    // Signing the transaction.
    SignedTransaction signedTx = getServiceHub().
    ↪signInitialTransaction(txBuilder);

    // Creating a session with the other party.
    FlowSession otherPartySession = initiateFlow(otherParty);

    // We finalise the transaction and then send it to the counterparty.
    subFlow(new FinalityFlow(signedTx, otherPartySession));

    return null;
}
}

```

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 is part of a flow pair and that it triggers the other side to run the the counterpart flow (which in our case is the `IOUFlowResponder` defined below).
- `@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 get the notary to sign it, have that recorded locally and then send it to all the relevant parties. Once that happens the transaction will become a permanent part of the ledger. We use `FinalityFlow` which does all of this for the lender.

For the borrower to receive the transaction they just need a flow that responds to the seller's.

## **Creating the borrower's flow**

The borrower has to use `ReceiveFinalityFlow` in order to receive and record the transaction; it needs to respond to the lender's flow. Let's do that by replacing `Responder` from the template with the following:

```
// Replace Responder's definition with:
@InitiatedBy(IOUflow::class)
class IOUflowResponder(private val otherPartySession: FlowSession) : FlowLogic<Unit>
    () {
    @Suspendable
    override fun call() {
        subFlow(ReceiveFinalityFlow(otherPartySession))
    }
}
```

```
// Replace Responder's definition with:
@InitiatedBy(IOUflow.class)
public class IOUflowResponder extends FlowLogic<Void> {
    private final FlowSession otherPartySession;

    public IOUflowResponder(FlowSession otherPartySession) {
        this.otherPartySession = otherPartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
        subFlow(new ReceiveFinalityFlow(otherPartySession));
    }
}
```

(continues on next page)

(continued from previous page)

```

        return null;
    }
}

```

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. This message will be the finalised transaction which will be recorded in the borrower's vault.

## 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.

### 9.1.4 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']) {

    nodeDefaults {
        cordapps = [
            "net.corda:corda-finance-contracts:$corda_release_version",
            "net.corda:corda-finance-workflows:$corda_release_version",
            "net.corda:corda-confidential-identities:$corda_release_version"
        ]
    }

    directory "./build/nodes"
    node {
        name "O=Notary,L=London,C=GB"
        notary = [validating : true]
        p2pPort 10002
        rpcPort 10003
    }
    node {
        name "O=PartyA,L=London,C=GB"
        p2pPort 10005
        rpcPort 10006
        webPort 10007
        rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL"] ]]
    }
    node {

```

(continues on next page)

(continued from previous page)

```

        name "O=PartyB, L>New York, C=US"
        p2pPort 10008
        rpcPort 10009
        webPort 10010
        sshdPort 10024
        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
    ↵'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."

```

Corda logo
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>>> 

```

## 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 [Node shell](#) page.

---

We want to create an IOU of 99 with PartyB. We start the `IOUFlow` by typing:

```
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:
```

(continues on next page)

(continued from previous page)

```
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.

However, if we run the same command on the other node (the notary), we will see the following:

```
{
  "states" : [ ],
  "statesMetadata" : [ ],
  "totalStatesAvailable" : -1,
  "stateTypes" : "UNCONSUMED",
  "otherResults" : [ ]}
```

This is the result of Corda's privacy model. Because the notary was not involved in the transaction and had no need to see the data, the transaction was not distributed to them.

## 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 blockchain
- 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 could add unit tests, using the contract-test and flow-test frameworks
- We could 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.

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 applications that are installed on one or more Corda nodes, and that allow the node's operator to instruct their node to perform some new process - anything from issuing a debt instrument to making a restaurant booking.

### 9.1.5 Our use-case

We will write a CorDapp to model IOUs on the blockchain. Each IOU – short for “I O(we) (yo)U” – will record the fact that one node owes another node a certain amount. This simple CorDapp will showcase several key benefits of Corda as a blockchain platform:

- **Privacy** - Since IOUs represent sensitive information, we will be taking advantage of Corda's ability to only share ledger updates with other nodes on a need-to-know basis, instead of using a gossip protocol to share this information with every node on the network as you would with a traditional blockchain platform
- **Well-known identities** - Each Corda node has a well-known identity on the network. This allows us to write code in terms of real identities, rather than anonymous public keys
- **Re-use of existing, proven technologies** - We will be writing our CorDapp using standard Java. It will run on a Corda node, which is simply a Java process and runs on a regular Java machine (e.g. on your local machine or in the cloud). The nodes will store their data in a standard SQL database

CorDapps usually define at least three things:

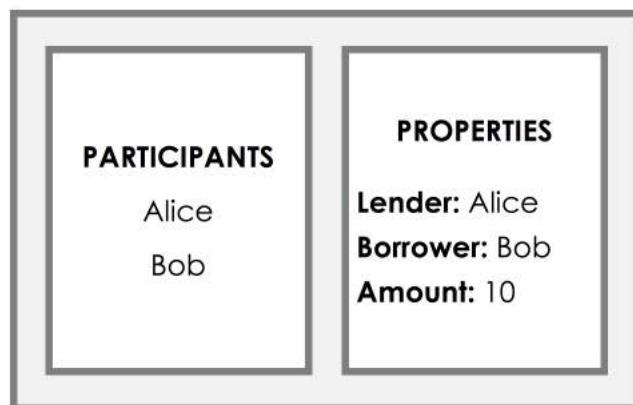
- **States** - the (possibly shared) facts that are written to the ledger
- **Flows** - the procedures for carrying out specific ledger updates
- **Contracts** - the constraints governing how states of a given type can evolve over time

Our IOU CorDapp is no exception. It will define the following components:

#### The `IOUState`

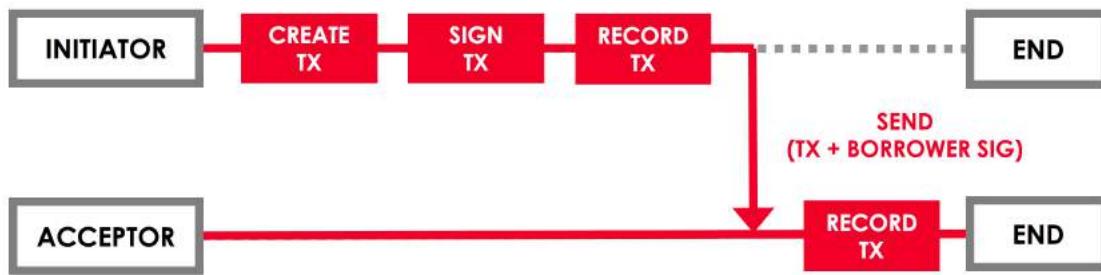
Our state will be the `IOUState`, representing an IOU. It will contain the IOU's value, its lender and its borrower. We can visualize `IOUState` as follows:

#### `IOU_STATE`



## The IOUFlow

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



## The IOUContract

For this tutorial, we will use the default `TemplateContract`. We will update it to create a fully-fledged `IOUContract` in the next tutorial.

### 9.1.6 Progress so far

We've designed a simple CorDapp that will allow nodes to agree new IOUs on the blockchain.

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

## 9.2 Hello, World! Pt.2 - Contract constraints

---

**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 blockchain
- An `IOUFlow` and `IOUFlowResponder` flow pair, 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 blockchain over time. Anyone was free to create IOUs of any value, between any party.

In this tutorial, we'll write a contract to imposes 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.

### 9.2.1 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:

```
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)

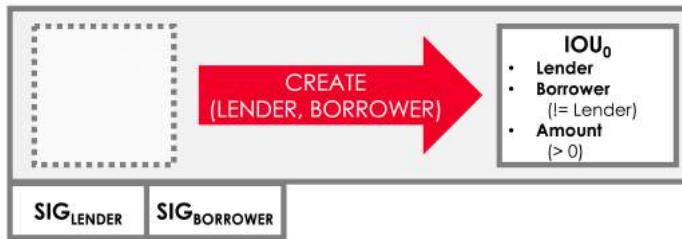
We might also want to impose some constraints on the properties of the issued `IOUState`:

- 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 `TemplateContract.kt` and updating to define an `IOUContract`:

```

// Add this import:
import net.corda.core.contracts.*

class IOUContract : Contract {
    companion object {
        const 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 output = tx.outputsOfType<IOUState>().single()
            "The IOU's value must be non-negative." using (output.value > 0)
            "The lender and the borrower cannot be the same entity." using (output.
            lender != output.borrower)
        }
    }
}
  
```

```

// Add these imports:
import net.corda.core.contracts.CommandWithParties;
import net.corda.core.identity.Party;
  
```

(continues on next page)

(continued from previous page)

```

import java.security.PublicKey;
import java.util.Arrays;
import java.util.List;

import static net.corda.core.contracts.ContractsDSL.requireSingleCommand;

// Replace TemplateContract's definition with:
public class IOUContract implements Contract {
    public static final String ID = "com.template.IOUContract";

    // Our Create command.
    public static class Create implements CommandData {
    }

    @Override
    public void verify(LedgerTransaction tx) {
        final CommandWithParties<IOUContract.Create> command =
            requireSingleCommand(tx.getCommands(), IOUContract.Create.class);

        // Constraints on the shape of the transaction.
        if (!tx.getInputs().isEmpty())
            throw new IllegalArgumentException("No inputs should be consumed when
issuing an IOU.");
        if (!(tx.getOutputs().size() == 1))
            throw new IllegalArgumentException("There should be one output state of
type IOUState.");

        // IOU-specific constraints.
        final IOUState output = tx.outputsOfType(IOUState.class).get(0);
        final Party lender = output.getLender();
        final Party borrower = output.getBorrower();
        if (output.getValue() <= 0)
            throw new IllegalArgumentException("The IOU's value must be non-negative.
");
        if (lender.equals(borrower))
            throw new IllegalArgumentException("The lender and the borrower cannot be
the same entity.");

        // Constraints on the signers.
        final List<PublicKey> requiredSigners = command.getSigners();
        final List<PublicKey> expectedSigners = Arrays.asList(borrower.getOwningKey(),
lender.getOwningKey());
        if (requiredSigners.size() != 2)
            throw new IllegalArgumentException("There must be two signers.");
        if (!(requiredSigners.containsAll(expectedSigners)))
            throw new IllegalArgumentException("The borrower and lender must be
signers.");
    }
}

```

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):

```
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.

In Kotlin, we impose these and the subsequent constraints 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.

In Java, we simply throw an `IllegalArgumentException` manually instead.

## 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

You can see that we're not restricted to only writing constraints inside `verify`. 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 blockchain 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.

### 9.2.2 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/Flows.kt, change the imports block to the following:

```
import co.paralleluniverse.fibers.Suspendable
import net.corda.core.contracts.Command
import net.corda.core.flows.CollectSignaturesFlow
import net.corda.core.flows.FinalityFlow
import net.corda.core.flows.FlowLogic
import net.corda.core.flows.InitiatingFlow
import net.corda.core.flows.StartableByRPC
import net.corda.core.identity.Party
import net.corda.core.transactions.TransactionBuilder
import net.corda.core.utilities.ProgressTracker
```

```
import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.contracts.Command;
import net.corda.core.flows.*;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.security.PublicKey;
import java.util.Arrays;
import java.util.List;
```

And update IOUFlow.call to the following:

```
// 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 command = Command(IOUSContract.Create(), listOf(ourIdentity.owningKey, otherParty.
    →owningKey))

// We create a transaction builder and add the components.
val txBuilder = TransactionBuilder(notary = notary)
    .addOutputState(outputState, IOUNContract.ID)
    .addCommand(command)

// 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()))
```

(continues on next page)

(continued from previous page)

```

// Finalising the transaction.
subFlow(FinalityFlow(fullySignedTx, otherPartySession))

// We retrieve the notary identity from the network map.
Party notary = getServiceHub().getNetworkMapCache().getNotaryIdentities().get(0);

// We create the transaction components.
IOUState outputState = new IOUState(iouValue, getOurIdentity(), otherParty);
List<PublicKey> requiredSigners = Arrays.asList(getOurIdentity().getOwningKey(), ↳
    ↳otherParty.getOwningKey());
Command command = new Command<>(new IOUContract.Create(), requiredSigners);

// We create a transaction builder and add the components.
TransactionBuilder txBuilder = new TransactionBuilder(notary)
    .addOutputState(outputState, IOUContract.ID)
    .addCommand(command);

// Verifying the transaction.
txBuilder.verify(getServiceHub());

// Signing the transaction.
SignedTransaction signedTx = getServiceHub().signInitialTransaction(txBuilder);

// Creating a session with the other party.
FlowSession otherPartySession = initiateFlow(otherParty);

// Obtaining the counterparty's signature.
SignedTransaction fullySignedTx = subFlow(new CollectSignaturesFlow(
    signedTx, Arrays.asList(otherPartySession), CollectSignaturesFlow.tracker()));

// Finalising the transaction.
subFlow(new FinalityFlow(fullySignedTx, otherPartySession));

return null;

```

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

Previously we wrote a responder flow for the borrower in order to receive the finalised transaction from the lender. We use this same flow to first request their signature over the transaction.

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`.

## Updating the borrower's flow

On the lender's side, we used `CollectSignaturesFlow` to automate the collection of signatures. To allow the borrower to respond, we need to update its responder flow to first receive the partially signed transaction for signing. Update `IOUFlowResponder.call` to be the following:

```
@Suspendable
override fun call() {
    val signTransactionFlow = object : SignTransactionFlow(otherPartySession) {
        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)
        }
    }

    val expectedTxId = subFlow(signTransactionFlow).id

    subFlow(ReceiveFinalityFlow(otherPartySession, expectedTxId))
}
```

```
@Suspendable
@Override
public Void call() throws FlowException {
    class SignTxFlow extends SignTransactionFlow {
        private SignTxFlow(FlowSession otherPartySession) {
            super(otherPartySession);
        }

        @Override
        protected void checkTransaction(SignedTransaction stx) {
            requireThat(require -> {
                ContractState output = stx.getTx().getOutputs().get(0).getData();
                require.using("This must be an IOU transaction.", output instanceof
                    IOUState);
                IOUState iou = (IOUState) output;
                require.using("The IOU's value can't be too high.", iou.getValue() <
                    100);
                return null;
            });
        }
    }

    SecureHash expectedTxId = subFlow(new SignTxFlow(otherPartySession)).getId();

    subFlow(new ReceiveFinalityFlow(otherPartySession, expectedTxId));

    return null;
}
```

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.

`SignedTransactionFlow` returns the newly signed transaction. We pass in the transaction's ID to `ReceiveFinalityFlow` to ensure we are recording the correct notarised transaction from the lender.

## 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 blockchain. 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](#).

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.

## 9.3 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:

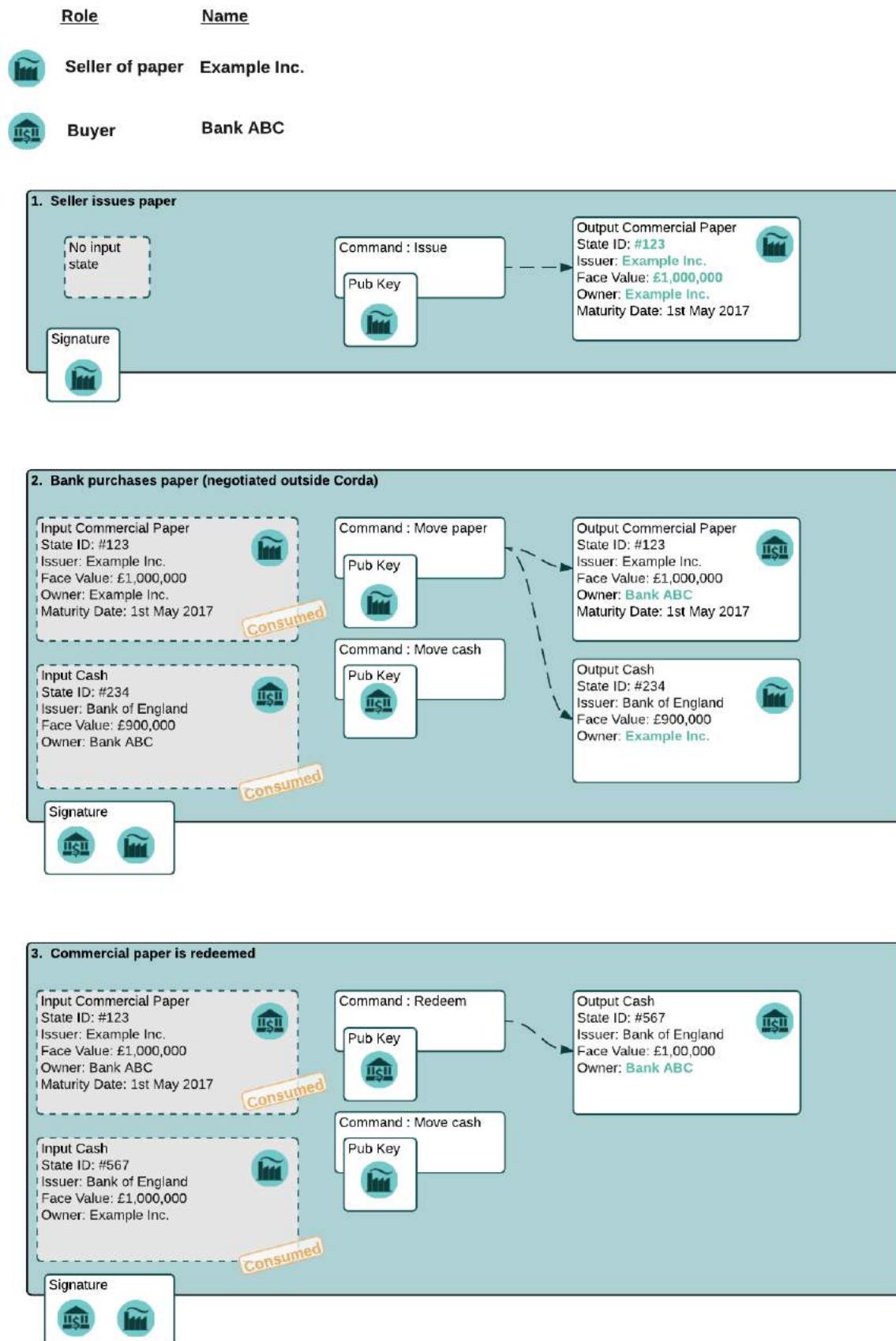
- Executable code (validation logic)
- State objects

- 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:



### 9.3.1 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.

```
class CommercialPaper : Contract {
    override fun verify(tx: LedgerTransaction) {
        TODO()
    }
}
```

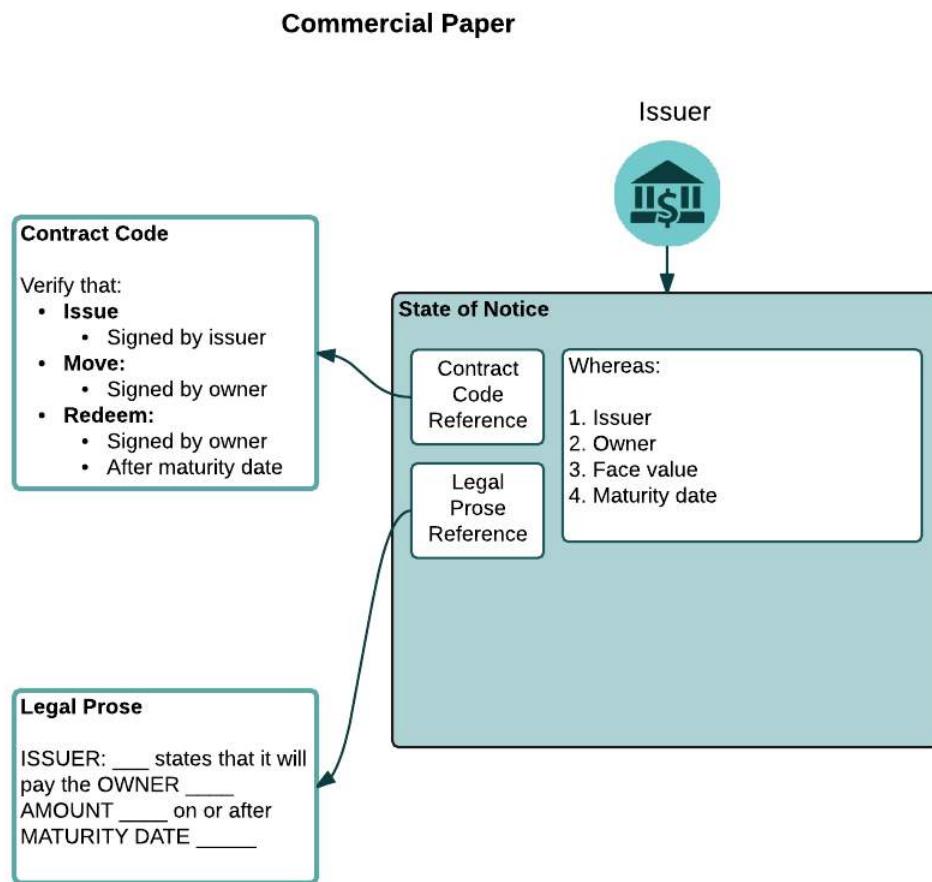
```
public class CommercialPaper implements Contract {
    @Override
    public void verify(LedgerTransaction tx) {
        throw new UnsupportedOperationException();
    }
}
```

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.

### 9.3.2 States

A state is a class that stores data that is checked by the contract. A commercial paper state is structured as below:



```

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))
}

```

```

public class State implements OwnableState {
    private PartyAndReference issuance;
    private AbstractParty owner;
    private Amount<Issued<Currency>> faceValue;
    private Instant maturityDate;

    public State() {
    } // For serialization
}

```

(continues on next page)

(continued from previous page)

```

public State(PartyAndReference issuance, AbstractParty owner, Amount<Issued
↪<Currency>> faceValue,
            Instant maturityDate) {
    this.issuance = issuance;
    this.owner = owner;
    this.faceValue = faceValue;
    this.maturityDate = maturityDate;
}

public State copy() {
    return new State(this.issuance, this.owner, this.faceValue, this.
↪maturityDate);
}

public State withoutOwner() {
    return new State(this.issuance, new AnonymousParty(NullKeys.NullPublicKey.
↪INSTANCE), this.faceValue, this.maturityDate);
}

@NotNull
@Override
public CommandAndState withNewOwner(@NotNull AbstractParty newOwner) {
    return new CommandAndState(new CommercialPaper.Commands.Move(), new_
↪State(this.issuance, newOwner, this.faceValue, this.maturityDate));
}

public PartyAndReference getIssuance() {
    return issuance;
}

public AbstractParty getOwner() {
    return owner;
}

public Amount<Issued<Currency>> getFaceValue() {
    return faceValue;
}

public Instant getMaturityDate() {
    return maturityDate;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    State state = (State) o;

    if (issuance != null ? !issuance.equals(state.issuance) : state.issuance !=_
↪null) return false;
    if (owner != null ? !owner.equals(state.owner) : state.owner != null) return_
↪false;
    if (faceValue != null ? !faceValue.equals(state.faceValue) : state.faceValue !_
↪= null) return false;
    return !(maturityDate != null ? !maturityDate.equals(state.maturityDate) :_
↪state.maturityDate != null);
}

```

(continues on next page)

(continued from previous page)

```

    }

    @Override
    public int hashCode() {
        int result = issuance != null ? issuance.hashCode() : 0;
        result = 31 * result + (owner != null ? owner.hashCode() : 0);
        result = 31 * result + (faceValue != null ? faceValue.hashCode() : 0);
        result = 31 * result + (maturityDate != null ? maturityDate.hashCode() : 0);
        return result;
    }

    @NotNull
    @Override
    public List<AbstractParty> getParticipants() {
        return ImmutableList.of(this.owner);
    }
}

```

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.

### 9.3.3 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:

```
interface Commands : CommandData {
    class Move : TypeOnlyCommandData(), Commands
    class Redeem : TypeOnlyCommandData(), Commands
    class Issue : TypeOnlyCommandData(), Commands
}
```

```
public static class Commands implements CommandData {
    public static class Move extends Commands {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof Move;
        }
    }

    public static class Redeem extends Commands {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof Redeem;
        }
    }

    public static class Issue extends Commands {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof Issue;
        }
    }
}
```

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.

### 9.3.4 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 transaction 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.

```
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>()
```

```

@Override
public void verify(LedgerTransaction tx) {
    List<InOutGroup<State, State>> groups = tx.groupStates(State.class,
        State::withoutOwner);
    CommandWithParties<Commands> cmd = requireSingleCommand(tx.getCommands(),
        Commands.class);
}

```

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.

### 9.3.5 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:

```
// 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

    ...
}
```

```
List<InOutGroup<State, Pair<PartyReference, Currency>>> groups = tx.groupStates(Cash.
    ↪State.class, s -> Pair(s.deposit, s.amount.currency))
for (InOutGroup<State, Pair<PartyReference, Currency>> group : groups) {
    List<State> inputs = group.getInputs();
    List<State> outputs = group.getOutputs();
    Pair<PartyReference, Currency> key = group.getKey();

    ...
}
```

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”:

```
val groups = tx.groupStates(State::withoutOwner)
```

```
List<InOutGroup<State, State>> groups = tx.groupStates(State.class,
    ↪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`.

### 9.3.6 Checking the requirements

After extracting the command and the groups, we then iterate over each group and verify it meets the required business logic.

```
val timeWindow: TimeWindow? = tx.timeWindow

for ((inputs, outputs, _) in groups) {
    when (command.value) {
        is Commands.Move -> {
            val input = inputs.single()
```

(continues on next page)

(continued from previous page)

```

        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.Redem -> {
        // 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")
}
}

```

```

TimeWindow timeWindow = tx.getTimeWindow();

for (InOutGroup group : groups) {
    List<State> inputs = group.getInputs();
    List<State> outputs = group.getOutputs();

    if (cmd.getValue() instanceof Commands.Move) {
        State input = inputs.get(0);

```

(continues on next page)

(continued from previous page)

```

        requireThat(require -> {
            require.using("the transaction is signed by the owner of the CP", cmd.
            ↪getSigners().contains(input.getOwner().getOwningKey())));
            require.using("the state is propagated", 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.
            return null;
        });

    } else if (cmd.getValue() instanceof Commands.Redeem) {
        // Redemption of the paper requires movement of on-ledger cash.
        State input = inputs.get(0);
        Amount<Issued<Currency>> received = sumCashBy(tx.getOutputStates(), input.
        ↪getOwner());
        if (timeWindow == null) throw new IllegalArgumentException("Redemptions must
        ↪be timestamped");
        Instant time = timeWindow.getFromTime();
        requireThat(require -> {
            require.using("the paper must have matured", time.isAfter(input.
            ↪getMaturityDate()));
            require.using("the received amount equals the face value", received ==
            ↪input.getFaceValue());
            require.using("the paper must be destroyed", outputs.isEmpty());
            require.using("the transaction is signed by the owner of the CP", cmd.
            ↪getSigners().contains(input.getOwner().getOwningKey()));
            return null;
        });
    } else if (cmd.getValue() instanceof Commands.Issue) {
        State output = outputs.get(0);
        if (timeWindow == null) throw new IllegalArgumentException("Issuances must
        ↪have a time-window");
        Instant time = timeWindow.getUntilTime();
        requireThat(require -> {
            // Don't allow people to issue commercial paper under other entities
            ↪identities.
            require.using("output states are issued by a command signer", cmd.
            ↪getSigners().contains(output.getIssuance().getParty().getOwningKey()));
            require.using("output values sum to more than the inputs", output.
            ↪getFaceValue().getQuantity() > 0);
            require.using("the maturity date is not in the past", time.
            ↪isBefore(output.getMaturityDate()));
            // Don't allow an existing CP state to be replaced by this issuance.
            require.using("can't reissue an existing state", inputs.isEmpty());
            return null;
        });
    } else {
        throw new IllegalArgumentException("Unrecognised command");
    }
}

```

This loop is the core logic of the contract.

The first line simply gets the time-window out of the transaction. Setting a time-window in 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 interoperability with Java, 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 time-window 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 bytecode.

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.

### 9.3.7 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](#).

### 9.3.8 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:

```
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

```
companion object {
    const val CP_PROGRAM_ID: ContractClassName = "net.corda.finance.contracts.
    ↪CommercialPaper"
}
```

```
public static final String IOU_CONTRACT_ID = "com.example.contract.IOUContract";
```

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?

```
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(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.

```
@Throws(InsufficientBalanceException::class)
fun generateRedeem(tx: TransactionBuilder, paper: StateAndRef<State>, services: ↵
    ServiceHub) {
    // Add the cash movement using the states in our vault.
    CashUtils.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(Command.Commands.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.

### 9.3.9 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](#).

### 9.3.10 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.

### 9.3.11 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](#)”.

### 9.3.12 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:

```
val encumbrance: Int? get() = null
```

```
@Nullable
@Override
public Integer getEncumbrance() {
    return null;
}
```

The time-lock contract mentioned above can be implemented very simply:

```

class TestTimeLock : Contract {
    ...
    override fun verify(tx: LedgerTransaction) {
        val time = tx.timeWindow?.untilTime ?: 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:

```

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.

## 9.4 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.

### 9.4.1 Setting up the test

Before writing the individual tests, the general test setup must be configured:

```

class CommercialPaperTest {
    private val miniCorp = TestIdentity(CordaX500Name("MiniCorp", "London", "GB"))
    private val megaCorp = TestIdentity(CordaX500Name("MegaCorp", "London", "GB"))
    private val ledgerServices = MockServices(listOf("net.corda.finance.schemas"),
        megaCorp, miniCorp)
    ...
}

```

```
public class CommercialPaperTest {
    private TestIdentity megaCorp = new TestIdentity(new CordaX500Name("MegaCorp",
    "London", "GB"));
    private TestIdentity miniCorp = new TestIdentity(new CordaX500Name("MiniCorp",
    "London", "GB"));
    MockServices ledgerServices = new MockServices(Arrays.asList("net.corda.finance.
    schemas"), megaCorp, miniCorp);
    ...
}
```

The `ledgerServices` object will provide configuration to the `ledger` DSL in the individual tests.

### 9.4.2 Testing single transactions

We start with the empty ledger:

```
class CommercialPaperTest {
    @Test
    fun emptyLedger() {
        ledgerServices.ledger {
            ...
        }
    }
}
```

```
public class CommercialPaperTest {
    @Test
    public void emptyLedger() {
        ledger(ledgerServices, l -> {
            ...
            return null;
        });
    }
}
```

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:

```
val bigCorp = TestIdentity((CordaX500Name("BigCorp", "New York", "GB")))
```

```
private static final TestIdentity bigCorp = new TestIdentity(new CordaX500Name(
    "BigCorp", "New York", "GB"));
```

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:

```
@Test
fun simpleCPDoesntCompile() {
    val inState = getPaper()
    ledger {
        transaction {
            input(CommercialPaper.CP_PROGRAM_ID) { inState }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

@Test
public void simpleCPDoesntCompile() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.input(inState);
        });
        return Unit.INSTANCE;
    });
}

```

We can add a transaction to the ledger using the `transaction` primitive. The transaction in turn may be defined by specifying inputs, outputs, commands and attachments.

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:

```
Error: (29, 17) Kotlin: Type mismatch: inferred type is Unit but EnforceVerifyOrFail
↳ was expected
```

```
Error: (35, 27) java: incompatible types: bad return type in lambda expression missing
↳ return value
```

This is deliberate: The DSL forces us to specify either `verifies()` or `fails with` ("some text") on the last line of `transaction`:

```
// 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()
        }
    }
}
```

```
// This example test will fail with this exception.
@Test(expected = IllegalStateException.class)
public void simpleCP() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.attachments(JCP_PROGRAM_ID);
            tx.input(JCP_PROGRAM_ID, inState);
            return tx.verifies();
        });
        return Unit.INSTANCE;
    });
}
```

(continues on next page)

(continued from previous page)

```
});  
}
```

Let's take a look at a transaction that fails.

```
// 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()  
        }  
    }  
}
```

```
// This example test will fail with this exception.  
@Test(expected = TransactionVerificationException.ContractRejection.class)  
public void simpleCPMove() {  
    ICommercialPaperState inState = getPaper();  
    ledger(ledgerServices, l -> {  
        l.transaction(tx -> {  
            tx.input(JCP_PROGRAM_ID, inState);  
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.  
                Move());  
            tx.attachments(JCP_PROGRAM_ID);  
            return tx.verifies();  
        });  
        return Unit.INSTANCE;  
    });  
}
```

When run, that code produces the following error:

```
net.corda.core.contracts.TransactionVerificationException$ContractRejection: java.  
lang.IllegalArgumentException: Failed requirement: the state is propagated
```

```
net.corda.core.contracts.TransactionVerificationException$ContractRejection: java.  
lang.IllegalStateException: 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"):

```
@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")  
        }  
    }  
}
```

(continues on next page)

(continued from previous page)

```

        }
    }
}
```

```

@Test
public void simpleCPMoveFails() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.input(JCP_PROGRAM_ID, inState);
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
                    Move());
            tx.attachments(JCP_PROGRAM_ID);
            return txfailsWith("the state is propagated");
        });
        return Unit.INSTANCE;
    });
}
```

We can continue to build the transaction until it verifies:

```

@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()
        }
    }
}
```

```

@Test
public void simpleCPMoveSuccessAndFailure() {
    ICommercialPaperState inState = getPaper();
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.input(JCP_PROGRAM_ID, inState);
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
                    Move());
            tx.attachments(JCP_PROGRAM_ID);
            tx.failsWith("the state is propagated");
            tx.output(JCP_PROGRAM_ID, "alice's paper", inState.withOwner(alice.
                    getParty()));
            return tx.verifies();
        });
        return Unit.INSTANCE;
    });
}
```

`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`:

```
@Test
fun `simple issuance with tweak`() {
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            output(JCP_PROGRAM_ID, "paper", getPaper()) // Some CP is issued onto the
            ↪ ledger by MegaCorp.
            attachments(JCP_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()
        }
    }
}
```

```
@Test
public void simpleIssuanceWithTweak() {
    ledger(ledgerServices, l -> {
        l.transaction(tx -> {
            tx.output(JCP_PROGRAM_ID, "paper", getPaper()); // Some CP is issued onto
            ↪ the ledger by MegaCorp.
            tx.attachments(JCP_PROGRAM_ID);
            tx.tweak(tw -> {
                tw.command(bigCorp.getPublicKey(), new JavaCommercialPaper.Commands.
            ↪ Issue());
                tw.timeWindow(TEST_TX_TIME);
                return tw.failsWith("output states are issued by a command signer");
            });
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
            ↪ Issue());
            tx.timeWindow(TEST_TX_TIME);
            return tx.verifies();
        });
        return Unit.INSTANCE;
    });
}
```

`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:

```
@Test
fun `simple issuance with tweak and top level transaction`() {
    ledgerServices.transaction(dummyNotary.party) {
```

(continues on next page)

(continued from previous page)

```

    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()
}
}

```

```

@Test
public void simpleIssuanceWithTweakTopLevelTx() {
    transaction(ledgerServices, tx -> {
        tx.output(JCP_PROGRAM_ID, "paper", getPaper()); // Some CP is issued onto the ledger by MegaCorp.
        tx.attachments(JCP_PROGRAM_ID);
        tx.tweak(tw -> {
            tw.command(bigCorp.getPublicKey(), new JavaCommercialPaper.Commands.Issue());
            tw.timeWindow(TEST_TX_TIME);
            return twfailsWith("output states are issued by a command signer");
        });
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.Issue());
        tx.timeWindow(TEST_TX_TIME);
        return tx.verifies();
    });
}

```

### 9.4.3 Chaining transactions

Now that we know how to define a single transaction, let's look at how to define a chain of them:

```

@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)
        }

        // 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()
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        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
            ↪<IClaimsState>().withOwner(alice.party))
            command(alice.publicKey, Cash.Commands.Move())
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }
    }
}

```

```

@Test
public void chainCommercialPaper() {
    PartyAndReference issuer = megaCorp.ref(defaultRef);
    ledgerServices, l -> {
        l.unverifiedTransaction(tx -> {
            tx.output(Cash.PROGRAM_ID, "alice's $900",
                new Cash.State(issuedBy(DOLLARS(900), issuer), alice.getParty()));
            tx.attachments(Cash.PROGRAM_ID);
            return Unit.INSTANCE;
        });

        // Some CP is issued onto the ledger by MegaCorp.
        l.transaction("Issuance", tx -> {
            tx.output(JCP_PROGRAM_ID, "paper", getPaper());
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
            ↪Issue());
            tx.attachments(JCP_PROGRAM_ID);
            tx.timeWindow(TEST_TX_TIME);
            return tx.verifies();
        });

        l.transaction("Trade", tx -> {
            tx.input("paper");
            tx.input("alice's $900");
            tx.output(Cash.PROGRAM_ID, "borrowed $900", new Cash.
            ↪State(issuedBy(DOLLARS(900), issuer), megaCorp.getParty()));
            JavaCommercialPaper.State inputPaper = l.
            ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
            tx.output(JCP_PROGRAM_ID, "alice's paper", inputPaper.withOwner(alice.
            ↪getParty()));
            tx.command(alice.getPublicKey(), new Cash.Commands.Move());
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
            ↪Move());
            return tx.verifies();
        });
        return Unit.INSTANCE;
    });
}

```

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<IClaimState>().

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:

```
@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
→<IClaimState>().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<IClaimState>
→().withOwner(bob.party))
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }

        fails()
    }
}
```

```
@Test
public void chainCommercialPaperDoubleSpend() {
    PartyAndReference issuer = megaCorp.ref(defaultRef);
    ledger(ledgerServices, l -> {
```

(continues on next page)

(continued from previous page)

```

    l.unverifiedTransaction(tx -> {
        tx.output(Cash.PROGRAM_ID, "alice's $900",
            new Cash.State(issuedBy(DOLLARS(900), issuer), alice.getParty())));
        tx.attachments(Cash.PROGRAM_ID);
        return Unit.INSTANCE;
    });

    // Some CP is issued onto the ledger by MegaCorp.
    l.transaction("Issuance", tx -> {
        tx.output(JCP_PROGRAM_ID, "paper", getPaper());
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Issue());
        tx.attachments(JCP_PROGRAM_ID);
        tx.timeWindow(TEST_TX_TIME);
        return tx.verifies();
    });

    l.transaction("Trade", tx -> {
        tx.input("paper");
        tx.input("alice's $900");
        tx.output(Cash.PROGRAM_ID, "borrowed $900", new Cash.
        ↪State(issuedBy(DOLLARS(900), issuer), megaCorp.getParty()));
        JavaCommercialPaper.State inputPaper = l.
        ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
        tx.output(JCP_PROGRAM_ID, "alice's paper", inputPaper.withOwner(alice.
        ↪getParty()));
        tx.command(alice.getPublicKey(), new Cash.Commands.Move());
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Move());
        return tx.verifies();
    });

    l.transaction(tx -> {
        tx.input("paper");
        JavaCommercialPaper.State inputPaper = l.
        ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
        // We moved a paper to other pubkey.
        tx.output(JCP_PROGRAM_ID, "bob's paper", inputPaper.withOwner(bob.
        ↪getParty()));
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Move());
        return tx.verifies();
    });
    lfails();
    return Unit.INSTANCE;
});
}

```

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:

```

@Test
fun `chain commercial tweak`() {
    val issuer = megaCorp.party.ref(123)
    ledgerServices.ledger(dummyNotary.party) {

```

(continues on next page)

(continued from previous page)

```

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(CommercialPaper::class.java))
    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()
}
}

```

```

@Test
public void chainCommercialPaperTweak() {
    PartyAndReference issuer = megaCorp.ref(defaultRef);
    ledger(l -> {
        l.unverifiedTransaction(tx -> {
            tx.output(Cash.PROGRAM_ID, "alice's $900",
                new Cash.State(issuedBy(DOLLARS(900), issuer), alice.getParty()));
            tx.attachments(Cash.PROGRAM_ID);
            return Unit.INSTANCE;
        });
    });

    // Some CP is issued onto the ledger by MegaCorp.
    l.transaction("Issuance", tx -> {

```

(continues on next page)

(continued from previous page)

```

        tx.output(JCP_PROGRAM_ID, "paper", getPaper());
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Issue());
        tx.attachments(JCP_PROGRAM_ID);
        tx.timeWindow(TEST_TX_TIME);
        return tx.verifies();
    });

    l.transaction("Trade", tx -> {
        tx.input("paper");
        tx.input("alice's $900");
        tx.output(Cash.PROGRAM_ID, "borrowed $900", new Cash.
        ↪State(issuedBy(DOLLARS(900), issuer), megaCorp.getParty()));
        JavaCommercialPaper.State inputPaper = 1.
        ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
        tx.output(JCP_PROGRAM_ID, "alice's paper", inputPaper.withOwner(alice.
        ↪getParty()));
        tx.command(alice.getPublicKey(), new Cash.Commands.
        ↪Move(JavaCommercialPaper.class));
        tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
        ↪Move());
        return tx.verifies();
    });

    l.tweak(lw -> {
        lw.transaction(tx -> {
            tx.input("paper");
            JavaCommercialPaper.State inputPaper = 1.
            ↪retrieveOutput(JavaCommercialPaper.State.class, "paper");
            // We moved a paper to another pubkey.
            tx.output(JCP_PROGRAM_ID, "bob's paper", inputPaper.withOwner(bob.
            ↪getParty()));
            tx.command(megaCorp.getPublicKey(), new JavaCommercialPaper.Commands.
            ↪Move());
            return tx.verifies();
        });
        lwfails();
        return Unit.INSTANCE;
    });
    l.verifies();
    return Unit.INSTANCE;
});
}

```

## 9.5 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.

### 9.5.1 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

### 9.5.2 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? {
```

### 9.5.3 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.

### 9.5.4 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`:

1. 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.
    }
}
```

2. Bank A instructs its node to accept the contract upgrade to `DummyContractV2` for the contract state.

```
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 A 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.

```
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.

## 9.6 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 creates a local network where all the nodes see each other and provides safe shutting down of nodes in the background.

```
driver(DriverParameters(startNodesInProcess = true, cordappsForAllNodes = FINANCE_
    ↪CORDAPPS)) {
    val aliceUser = User("aliceUser", "testPassword1", permissions = setOf(
        startFlow<CashIssueAndPaymentFlow>(),
        invokeRpc("vaultTrackBy")
    ))

    val bobUser = User("bobUser", "testPassword2", permissions = setOf(
        startFlow<CashPaymentFlow>(),
        invokeRpc("vaultTrackBy")
    ))

    val (alice, bob) = listOf(
        startNode(providedName = ALICE_NAME, rpcUsers = listOf(aliceUser)),
        startNode(providedName = BOB_NAME, rpcUsers = listOf(bobUser))
    ).map { it.getOrThrow() }
}
```

```
driver(new DriverParameters()
    .withStartNodesInProcess(true)
    .withCordappsForAllNodes(FINANCE_CORDAPPS), dsl -> {

    User aliceUser = new User("aliceUser", "testPassword1", new HashSet<>(asList(
        startFlow(CashIssueAndPaymentFlow.class),
        invokeRpc("vaultTrack"))
    )
```

(continues on next page)

(continued from previous page)

```

        ));

        User bobUser = new User("bobUser", "testPassword2", new HashSet<>(asList(
            startFlow(CashPaymentFlow.class),
            invokeRpc("vaultTrack")
        ));

        try {
            List<CordaFuture<NodeHandle>> nodeHandleFutures = asList(
                dsl.startNode(new NodeParameters().withProvidedName(ALICE_NAME) .
            ↪withRpcUsers(singletonList(aliceUser))),
                dsl.startNode(new NodeParameters().withProvidedName(BOB_NAME) .
            ↪withRpcUsers(singletonList(bobUser)))
            );
        }

        NodeHandle alice = nodeHandleFutures.get(0).get();
        NodeHandle bob = nodeHandleFutures.get(1).get();
    }
}

```

The above code starts two nodes:

- Alice, configured with an RPC user who has permissions to start the `CashIssueAndPaymentFlow` flow on it and query Alice's vault.
- Bob, configured with an RPC user who only has permissions to start the `CashPaymentFlow` and query Bob's vault.

---

**Note:** You will notice that we did not start a notary. This is done automatically for us by the driver - it creates a notary node with the name `DUMMY_NOTARY_NAME` which is visible to both nodes. If you wish to customise this, for example create more notaries, then specify the `DriverParameters.notarySpecs` parameter.

---

The `startNode` function returns a `CordaFuture` object that completes once the node is fully started and visible on the local network. Returning a future allows starting of the nodes to be parallel. We wait on these futures as we need the information returned; their respective `NodeHandles`.

```

val aliceClient = CordaRPCClient(alice.rpcAddress)
val aliceProxy: CordaRPCOps = aliceClient.start("aliceUser", "testPassword1").proxy

val bobClient = CordaRPCClient(bob.rpcAddress)
val bobProxy: CordaRPCOps = bobClient.start("bobUser", "testPassword2").proxy

```

```

CordaRPCClient aliceClient = new CordaRPCClient(alice.getRpcAddress());
CordaRPCOps aliceProxy = aliceClient.start("aliceUser", "testPassword1").getProxy();

CordaRPCClient bobClient = new CordaRPCClient(bob.getRpcAddress());
CordaRPCOps bobProxy = bobClient.start("bobUser", "testPassword2").getProxy();

```

Next we connect to Alice and Bob from the test process using the test users we created. We establish RPC links that allow us to start flows and query state.

```

val bobVaultUpdates: Observable<Vault.Update<Cash.State>> = bobProxy.vaultTrackBy
    ↪<Cash.State>().updates
val aliceVaultUpdates: Observable<Vault.Update<Cash.State>> = aliceProxy.vaultTrackBy
    ↪<Cash.State>().updates

```

```
Observable<Vault.Update<Cash.State>> bobVaultUpdates = bobProxy.vaultTrack(Cash.State.
    ↪class).getUpdates();
Observable<Vault.Update<Cash.State>> aliceVaultUpdates = aliceProxy.vaultTrack(Cash.
    ↪State.class).getUpdates();
```

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)
aliceProxy.startFlow(::CashIssueAndPaymentFlow,
    1000.DOLLARS,
    issueRef,
    bob.nodeInfo.singleIdentity(),
    true,
    defaultNotaryIdentity
).returnValue.getOrThrow()

bobVaultUpdates.expectEvents {
    expect { update ->
        println("Bob got vault update of $update")
        val amount: Amount<Issued<Currency>> = update.produced.first().state.data.
    ↪amount
        assertEquals(1000.DOLLARS, amount.withoutIssuer())
    }
}
```

```
OpaqueBytes issueRef = OpaqueBytes.of((byte) 0);
aliceProxy.startFlowDynamic(
    CashIssueAndPaymentFlow.class,
    DOLLARS(1000),
    issueRef,
    bob.getNodeInfo().getLegalIdentities().get(0),
    true,
    dsl.getDefaultNotaryIdentity()
).getReturnValue().get();

@SuppressWarnings("unchecked")
Class<Vault.Update<Cash.State>> cashVaultUpdateClass = (Class<Vault.Update<Cash.State>
    ↪>) (Class<?>) Vault.Update.class;

expectEvents(bobVaultUpdates, true, () ->
    expect(cashVaultUpdateClass, update -> true, update -> {
        System.out.println("Bob got vault update of " + update);
        Amount<Issued<Currency>> amount = update.getProduced().iterator().next().
    ↪getState().getData().getAmount();
        assertEquals(DOLLARS(1000), Structures.withoutIssuer(amount));
        return null;
    })
);
```

We start a `CashIssueAndPaymentFlow` flow on the Alice node. We specify that we want Alice to self-issue \$1000 which is to be payed to Bob. We specify the default notary identity created by the driver 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 expect a single update to Bob's vault when it receives the \$1000 from Alice. This is what the `expectEvents` call is asserting.

```

bobProxy.startFlow(::CashPaymentFlow, 1000.DOLLARS, alice.nodeInfo.singleIdentity()).
    ↪returnValue.getOrThrow()

aliceVaultUpdates.expectEvents {
    expect { update ->
        println("Alice got vault update of $update")
        val amount: Amount<Issued<Currency>> = update.produced.first().state.data.
    ↪amount
        assertEquals(1000.DOLLARS, amount.withoutIssuer())
    }
}

```

```

bobProxy.startFlowDynamic(
    CashPaymentFlow.class,
    DOLLARS(1000),
    alice.getNodeInfo().getLegalIdentities().get(0)
).getReturnValue().get();

expectEvents(aliceVaultUpdates, true, () ->
    expect(cashVaultUpdateClass, update -> true, update -> {
        System.out.println("Alice got vault update of " + update);
        Amount<Issued<Currency>> amount = update.getProduced().iterator().next().
    ↪getState().getData().getAmount();
        assertEquals(DOLLARS(1000), Structures.withoutIssuer(amount));
        return null;
    })
);

```

Next we want Bob to send this cash back to Alice.

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 `CashIssueAndPaymentFlow` and `CashPaymentFlow`.

You can find the complete test at `example-code/src/integration-test/java/net/corda/docs/java/tutorial/test/JavaIntegrationTestingTutorial.java` (Java) and `example-code/src/integration-test/kotlin/net/corda/docs/kotlin/tutorial/test/KotlinIntegrationTestingTutorial.kt` (Kotlin) in the [Corda repo](#).

---

**Note:** To make sure the driver classes are included in your project you will need the following in your build.gradle file in the module in which you want to test:

```
testCompile "$corda_release_group:corda-node-driver:$corda_release_version"
```

---

## 9.7 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 [Interacting with a node](#).

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
}

@Suppress("DEPRECATION")
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, cordappsForAllNodes =
        ↪FINANCE_CORDAPPS, 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 = CordaRPCCClient(node.rpcAddress)
val proxy = client.start("user", "password").proxy

thread {
    generateTransactions(proxy)
}

```

`proxy` exposes the full RPC interface of the node:

```

/** 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.

```

(continues on next page)

(continued from previous page)

```

* 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> vaultQueryWithPagingSpec(contractStateType: Class<out T>
→, criteria: QueryCriteria, paging: PageSpecification): Vault.Page<T>

fun <T : ContractState> vaultQueryWithSorting(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.

```

(continues on next page)

(continued from previous page)

```

    *      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> vaultTrackWithPagingSpec(contractStateType: Class<out T>
    ↪, criteria: QueryCriteria, paging: PageSpecification): DataFeed<Vault.Page<T>,_
    ↪Vault.Update<T>>

        fun <T : ContractState> vaultTrackWithSorting(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 the full transaction for the provided ID
     *
     * TODO This method should be removed once SGX work is finalised and the design
     ↪of the corresponding API using [FilteredTransaction] can be started
     */
    @CordaInternal
    @Deprecated("This method is intended only for internal use and will be removed
    ↪from the public API soon.")
    fun internalFindVerifiedTransaction(txnId: SecureHash): 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
    
```

(continues on next page)

(continued from previous page)

```

    */
    @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 the network parameters the node is operating under. */
                val networkParameters: NetworkParameters

                /**
                 * 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.
                
```

(continues on next page)

(continued from previous page)

```

    * @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>

    /**
     * Attempts to kill a flow. This is not a clean termination and should be
     ↪reserved for exceptional cases such as stuck fibers.
     *
     * @return whether the flow existed and was killed.
     */
    fun killFlow(id: StateMachineRunId): Boolean

    /** 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. */

```

(continues on next page)

(continued from previous page)

```

fun openAttachment(id: SecureHash): InputStream

/** Uploads a jar to the node, returns its hash. */
@Throws(java.nio.file.FileAlreadyExistsException::class)
fun uploadAttachment(jar: InputStream): SecureHash

/** Uploads a jar including metadata to the node, returns its hash. */
@Throws(java.nio.file.FileAlreadyExistsException::class)
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 its [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?

/**

```

(continues on next page)

(continued from previous page)

```

    * 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. Notice that after invoking this method your node will lose
     * network map data and effectively won't be able to start any flow with the peers until network map is downloaded
     * again on next poll - from `additional-node-infos` directory or from network map server. It depends on the
     * polling interval when it happens. You can also use [refreshNetworkMapCache] to force next fetch from network map server
     * (not from directory - it will happen automatically).
     * If you run local test deployment and want clear view of the network, you may want to clear also `additional-node-infos` directory, because cache can be repopulated from there.
     */
    fun clearNetworkMapCache()

    /**
     * Poll network map server if available for the network map. Notice that you need to have `compatibilityZone`
     * or `networkServices` configured. This is normally done automatically on the regular time interval, but you may wish to
     * have the fresh view of network earlier.
     */
    fun refreshNetworkMapCache()

    /** 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.
     */

```

(continues on next page)

(continued from previous page)

```

    * @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

    /**
     * Shuts the node down. Returns immediately.
     * This does not wait for flows to be completed.
     */
fun shutdown()

    /**
     * Shuts the node down. Returns immediately.
     * @param drainPendingFlows whether the node will wait for pending flows to be
     * completed before exiting. While draining, new flows from RPC will be rejected.
     */
fun terminate(drainPendingFlows: Boolean = false)

    /**
     * Returns whether the node is waiting for pending flows to complete before
     * shutting down.
     * Disabling draining mode cancels this state.
     *
     * @return whether the node will shutdown when the pending flows count reaches
     * zero.
     */
fun isWaitingForShutdown(): 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 *Interacting with a node*

```
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}")
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        }
    }
    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.

### 9.7.1 Whitelisting classes from your CorDapp with the Corda node

As described in [Interacting with a node](#), 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](#).

### 9.7.2 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](#).

## 9.8 Building transactions

### 9.8.1 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.

### 9.8.2 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 time-window 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`. 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.

### 9.8.3 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` (in the [main Corda repo](#)). Second, a simple business model in which parties manually accept or reject each other's trade proposals, which is implemented in `WorkflowTransactionBuildTutorial.kt` (in the [main Corda repo](#)). 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 `UniqueIdentifier` 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()
```

## 9.8.4 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.

## 9.8.5 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)
```

(continues on next page)

(continued from previous page)

```

} else {
    listOf(transferredFundsOutput)
}
return Pair(inputs, outputs)

```

## 9.8.6 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)

```

(continues on next page)

(continued from previous page)

```

// 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())
}

```

## 9.8.7 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.
}

```

(continues on next page)

(continued from previous page)

```

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 for 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.

### 9.8.8 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, sourceSession))
```

### 9.8.9 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 [Transaction tear-offs](#).

## 9.9 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.

### 9.9.1 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.

There are many benefits of this flow based design and some development complexities as well. Some of the development challenges include:

- 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.

### 9.9.2 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.

### 9.9.3 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/workflows/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.

---

```
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>() { (continues on next page)
}
```

(continued from previous page)

```

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.

## 9.9.4 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.

---

## 9.9.5 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.

## 9.9.6 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 `startTrackedFlow` 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.

---

## 9.9.7 Implementing the seller

Let's implement the `Seller.call` method that will be run when the flow is invoked.

```
@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 subFlow(ReceiveFinalityFlow(otherSideSession, expectedTxId = 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 time-window in the transaction (if any) is valid and there are no double spends. In this flow, finalization is handled by the buyer, we just wait for them to send it to us. It will have the same ID as the one we started with but more signatures.

### 9.9.8 Implementing the buyer

OK, let's do the same for the buyer side:

```
@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))
```

(continues on next page)

(continued from previous page)

```

val twiceSignedTx = partSignedTx + sellerSignature
// DOCEND 6

// Notarise and record the transaction.
progressTracker.currentStep = RECORDING
return subFlow(FinalityFlow(twiceSignedTx, sellerSession))
}

@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) {"Well known_
↳identity lookup returned identity that does not match 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) = CashUtils.generateSpend(serviceHub, ptx,_
↳tradeRequest.price, ourIdentityAndCert, tradeRequest.payToIdentity.party)

    // Add inputs/outputs/a command for the movement of the asset.
    tx.addInputState(assetForSale)
}

```

(continues on next page)

(continued from previous page)

```

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.

### 9.9.9 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.

## 9.9.10 Sub-flows

Flows can be composed via nesting. Invoking a sub-flow looks similar to an ordinary function call:

```
@Suspendable
fun call() {
    val unnotarisedTransaction = ...
    subFlow(FinalityFlow(unnotarisedTransaction))
}
```

```
@Suspendable
public void call() throws FlowException {
    SignedTransaction unnotarisedTransaction = ...
    subFlow(new 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.

On the seller side we use `ReceiveFinalityFlow` to receive and record the finalised transaction.

**Warning:** If the buyer stops before sending the finalised transaction to the seller, the buyer is left with a valid transaction but the seller isn't, so they don't get the cash! This sort of thing is not always a risk (as the buyer 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:

```
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.

### 9.9.11 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.

---

**Warning:** If a node has flows still in a suspended state, with flow continuations written to disk, it will not be possible to upgrade that node to a new version of Corda or your app, because flows must be completely "drained" before an upgrade can be performed, and must reach a finished state for draining to complete (see [Draining the node](#) for details). While there are mechanisms for "evolving" serialised data held in the vault, there are no equivalent mechanisms for updating serialised checkpoint data. For this reason it is not a good idea to design flows with the intention that they should remain in a suspended state for a long period of time, as this will obstruct necessary upgrades to Corda itself. Any long-running business process should therefore be structured as a series of discrete transactions, written to the vault, rather than a single flow persisted over time through the flow checkpointing mechanism.

`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.

### 9.9.12 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.

### 9.9.13 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:

```
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)
```

```
private final ProgressTracker progressTracker = new ProgressTracker(
    RECEIVING,
    VERIFYING,
    SIGNING,
    COLLECTING_SIGNATURES,
    RECORDING
);

private static final ProgressTracker.Step RECEIVING = new ProgressTracker.Step(
    "Waiting for seller trading info");
private static final ProgressTracker.Step VERIFYING = new ProgressTracker.Step(
    "Verifying seller assets");
private static final ProgressTracker.Step SIGNING = new ProgressTracker.Step(
    "Generating and signing transaction proposal");
private static final ProgressTracker.Step COLLECTING_SIGNATURES = new ProgressTracker.
    ↪Step(
        "Collecting signatures from other parties");
```

(continues on next page)

(continued from previous page)

```
private static final ProgressTracker.Step RECORDING = new ProgressTracker.Step(
    "Recording completed transaction");
```

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:

```
object VERIFYING_AND_SIGNING : ProgressTracker.Step("Verifying and signing transaction proposal") {
    override fun childProgressTracker() = SignTransactionFlow.tracker()
}
```

```
private static final ProgressTracker.Step VERIFYING_AND_SIGNING = new ProgressTracker.Step("Verifying and signing transaction proposal") {
    @Nullable
    @Override
    public ProgressTracker childProgressTracker() {
        return SignTransactionFlow.Companion.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.

### 9.9.14 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

## 9.10 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: MockNetwork
    private lateinit var notaryNode: StartedMockNode
    private lateinit var megaCorpNode: StartedMockNode
    private lateinit var miniCorpNode: StartedMockNode
    private lateinit var megaCorp: Party
    private lateinit var miniCorp: Party
    private lateinit var notary: Party

    @Before
    fun setup() {
        mockNet = MockNetwork(MockNetworkParameters(cordappsForAllNodes =
    ↳ listOf(DUMMY_CONTRACTS_CORDAPP, enclosedCordapp())))
        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.startFlow(p)
    mockNet.runNetwork()
    future.getOrThrow()
    miniCorpNode.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)
            }
        }
    }
    megaCorpNode.transaction {
        megaCorpNode.services.recordTransactions(dummy1)
    }
    val dummy2: SignedTransaction = DummyContract.move(dummy1.tx.outRef(0), miniCorp).
    ↪let {
        val ptx = megaCorpNode.services.signInitialTransaction(it)
        notaryNode.services.addSignature(ptx, notary.owningKey)
    }
    megaCorpNode.transaction {
        megaCorpNode.services.recordTransactions(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.

## 9.11 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.

### 9.11.1 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.

## 9.11.2 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
  2. 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)
  3. 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)
            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.

### 9.11.3 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
```

(continues on next page)

(continued from previous page)

```

    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.

---

#### 9.11.4 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 TestCordapp objects which

point to CorDapps on the classpath. These CorDapps will be installed on each node on the network. Make sure the packages you provide reference to the CorDapp containing your oracle service.

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.

## 9.12 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.

The first step is to create a service class in your CorDapp that extends the `NotaryService` abstract class. This will ensure that it is recognised as a notary service. The custom notary service class should provide a constructor with two parameters of types `ServiceHubInternal` and `PublicKey`. Note that `ServiceHubInternal` does not provide any API stability guarantees.

```
class MyCustomValidatingNotaryService(override val services: ServiceHubInternal,
    ↪override val notaryIdentityKey: PublicKey) : SinglePartyNotaryService() {
    override val uniquenessProvider = PersistentUniquenessProvider(services.clock,
    ↪services.database, services.cacheFactory)

    override fun createServiceFlow(otherPartySession: FlowSession): FlowLogic<Void?> {
        ↪= MyValidatingNotaryFlow(otherPartySession, this)
    }
}
```

(continues on next page)

(continued from previous page)

```

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) : ValidatingNotaryFlow(otherSide, service) {
    override fun verifyTransaction(requestPayload: NotarisationPayload) {
        try {
            val stx = requestPayload.signedTransaction
            resolveAndContractVerify(stx)
            verifySignatures(stx)
            customVerify(stx)
        } catch (e: Exception) {
            throw NotaryInternalException(NotaryError.TransactionInvalid(e))
        }
    }

    @Suspendable
    private fun resolveAndContractVerify(stx: SignedTransaction) {
        subFlow(ResolveTransactionsFlow(stx, otherSideSession))
        stx.verify(serviceHub, false)
    }

    private fun verifySignatures(stx: SignedTransaction) {
        val transactionWithSignatures = stx.
        ↪resolveTransactionWithSignatures(serviceHub)
        checkSignatures(transactionWithSignatures)
    }

    private fun checkSignatures(tx: TransactionWithSignatures) {
        tx.verifySignaturesExcept(service.notaryIdentityKey)
    }

    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
    className : "net.corda.notarydemo.MyCustomValidatingNotaryService" # The fully-
    ↪qualified name of your service class
}

```

## 9.13 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.
    //
```

(continues on next page)

(continued from previous page)

```

    // 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`.

## 9.14 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 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 [Node shell](#).

It is encouraged that where possible attachments are reusable data, so that nodes can meaningfully cache them.

### 9.14.1 Uploading and downloading

To upload an attachment to the node, or download an attachment named by its hash, you use [Interacting with a node](#). 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: AB7FED7663A3F195A59A0F01091932B15C22405CB727A1518418BF53C6E666
```

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.

## 9.14.2 Searching for attachments

Attachment metadata can be queried in a similar way to the vault (see [API: Vault Query](#)).

`AttachmentQueryCriteria` can be used to build a query using the following set of column operations:

- 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)

The `and` and `or` operators can be used to build complex queries. For example:

```
rtEquals(
    emptyList(),
    storage.queryAttachments(
        AttachmentsQueryCriteria(uploaderCondition = Builder.equal("complexA"))
        .and(AttachmentsQueryCriteria(uploaderCondition = Builder.equal(
    ↵"complexB"))))

rtEquals(
    listOf(hashA, hashB),
    storage.queryAttachments(
        AttachmentsQueryCriteria(uploaderCondition = Builder.equal("complexA"))
        .or(AttachmentsQueryCriteria(uploaderCondition = Builder.equal(
    ↵"complexB"))))

complexCondition =
    (uploaderCondition("complexB").and(filenamerCondition("archiveB.zip")))
    ↵or(filenamerCondition("archiveC.zip"))
```

## 9.14.3 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.

---

#### 9.14.4 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)) { "attachment matching hash: ${state.hash} does not exist" }

            // 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 {
                while (true) {
                    val e = it.nextJarEntry ?: break
                    println("Entry > ${e.name}")
                    it.closeEntry()
                }
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        } 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)
    sender(rpc, inputStream, hash)
}

private fun sender(rpc: CordaRPCOps, inputStream: InputStream, hash: SecureHash.
    ↳SHA256) {
    // Get the identity key of the other side (the recipient).
    val notaryParty = rpc.partiesFromName("Notary", false).firstOrNull() ?: throw
    ↳IllegalArgumentException("Couldn't find notary party")
    val bankBParty = rpc.partiesFromName("Bank B", false).firstOrNull() ?: throw
    ↳IllegalArgumentException("Couldn't find Bank B party")
    // Make sure we have the file in storage
    if (!rpc.attachmentExists(hash)) {
        inputStream.use {
            val id = rpc.uploadAttachment(it)
            require(hash == id) { "Id was '$id' instead of '$hash'" }
        }
        require(rpc.attachmentExists(hash)) { "Attachment matching hash: $hash does
    ↳not exist" }
    }

    val flowHandle = rpc.startTrackedFlow(::AttachmentDemoFlow, bankBParty,
    ↳notaryParty, 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.

## 9.15 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.

### 9.15.1 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 described the next scheduled activity for itself. If it omits to do so, then nothing will be scheduled.

### 9.15.2 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`.

---

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.

### 9.15.3 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.

```
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.

## 9.16 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:

```
@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()
```

(continues on next page)

(continued from previous page)

```

        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.

### 9.16.1 Caveats

- By default, vault queries do not differentiate between states you recorded as a participant/owner, and states you recorded as an observer. You will have to write custom vault queries that only return states for which you are a participant/owner. See <https://docs.corda.net/api-vault-query.html#example-usage> for information on how to do this. This also means that Cash.generateSpend should not be used when recording Cash.State states as an observer
- 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>
- When an observer node is sent a transaction with the ALL\_VISIBLE flag set, any transactions in the transaction history that have not already been received will also have ALL\_VISIBLE states recorded. This means a node that is both an observer and a participant may have some transactions with all states recorded and some with

only relevant states recorded, even if those transactions are part of the same chain. As a result, there may be more states present in the vault than would be expected if just those transactions sent with the ALL\_VISIBLE recording flag were processed in this way.

## TOOLS

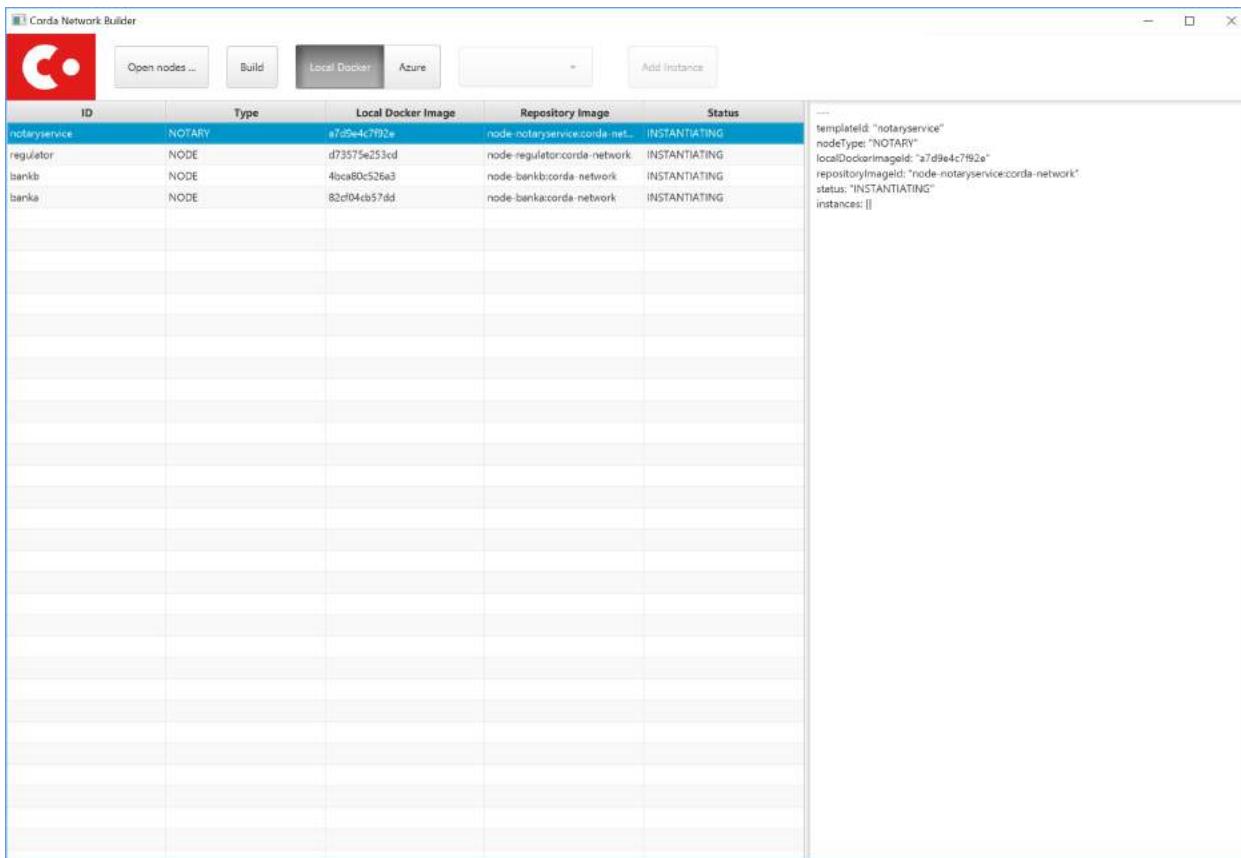
Corda provides various command line and GUI tools to help you as you work. Along with the three below, you may also wish to try the [Blob Inspector](#).

### 10.1 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.



The network you build will either be made up of local Docker nodes *or* of nodes spread across Azure containers. For each node a separate Docker image is built based on `corda/corda-zulu-4.1`. Unlike the official image, a `node.conf` file and CorDapps are embedded into the image (they are not externally provided to the running container via volumes/mount points). More backends may be added in future. The tool is open source, so contributions to add more destinations for the containers are welcome!

Download the Corda Network Builder.

### 10.1.1 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>)

### 10.1.2 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/samples`
2. `cd samples/cordapp-example`
3. `./gradlew clean workflows-java:deployNodes`

### 10.1.3 Building a network via the command line

#### Starting the nodes

##### Quickstart Local Docker

1. `cd workflows-java/build/nodes`
2. `java -jar <path/to/corda-tools-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			
		NAMES		
406868b4ba69	node-partyc:corda-network	"run-corda"	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	"run-corda"	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	"run-corda"	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	party0		
cf7ab689f493	node-notary:corda-network	"run-corda"	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		

Depending on your machine performance, even after all containers are reported as running, the underlying Corda nodes may be still starting and SSHing to a node may be not available immediately.

##### Quickstart Remote Azure

1. `cd kotlin-source/build/nodes`
2. `java -jar <path/to/corda-tools-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 `party0` 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": 4, "serial": 1532701330613 },
    {
        "addresses": [ "notary0:10020" ], "legalIdentitiesAndCerts": [ "O=Notary, L=London, C=GB" ], "platformVersion": 4, "serial": 1532701305115 },
    {
        "addresses": [ "partyC0:10020" ], "legalIdentitiesAndCerts": [ "O=PartyC, L=Paris, C=FR" ], "platformVersion": 4, "serial": 1532701331608 },
    {
        "addresses": [ "partyB0:10020" ], "legalIdentitiesAndCerts": [ "O=PartyB, L=New York, C=US" ], "platformVersion": 4, "serial": 1532701330118 }
]
>>>

```

You can also run a flow from cordapp-example: `flow start com.example.flow.ExampleFlow$Initiator iouValue: 20, otherParty: "PartyB"`

To verify it, connect into the partyb0 node and run `run vaultQuery contractStateType: "com.example.state.IOUState"`. The partyb0 vault should contain IOUState.

## 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 <path/to/corda-tools-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": 4, "serial": 1532701330613 },
    {
        "addresses": [ "notary0:10020" ], "legalIdentitiesAndCerts": [ "O=Notary, L=London, C=GB" ], "platformVersion": 4, "serial": 1532701305115 },
    {
        "addresses": [ "partyC0:10020" ], "legalIdentitiesAndCerts": [ "O=PartyC, L=Paris, C=FR" ], "platformVersion": 4, "serial": 1532701331608 },
    {
        "addresses": [ "partyB0:10020" ], "legalIdentitiesAndCerts": [ "O=PartyB, L=New York, C=US" ], "platformVersion": 4, "serial": 1532701330118 },
    {
        "addresses": [ "partyA1:10020" ], "legalIdentitiesAndCerts": [ "O=PartyZ, L=London, C=GB" ], "platformVersion": 4, "serial": 1532701630861 }
]
```

### 10.1.4 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/corda-tools-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			
		NAMES		
406868b4ba69	node-partyc:corda-network	"run-corda"	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	"run-corda"	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	"run-corda"	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	"run-corda"	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" : 4, "serial" : 1532701330613 },
  { "addresses" : [ "notary0:10020" ], "legalIdentitiesAndCerts" : [ "O=Notary, L=London, C=GB" ], "platformVersion" : 4, "serial" : 1532701305115 },
  { "addresses" : [ "partyc0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyC, L=Paris, C=FR" ], "platformVersion" : 4, "serial" : 1532701331608 },
  { "addresses" : [ "partyb0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyB, L>New York, C=US" ], "platformVersion" : 4, "serial" : 1532701330118 },
  { "addresses" : [ "partya1:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyZ, L=London, C=GB" ], "platformVersion" : 4, "serial" : 1532701630861 }
]

```

### 10.1.5 Shutting down the nodes

Run `docker kill $(docker ps -q)` to kill all running Docker processes.

## 10.2 Network Bootstrapper

### 10.2.1 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 [The network map](#).

### 10.2.2 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. “`devMode`” must be set to true. Then run the following command:

```
java -jar network-bootstrapper-4.1.jar --dir <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 Network 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.

### 10.2.3 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 `--copy-cordapps=No` option.

---

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` [The network map](#)).

By default the Bootstrapper will whitelist all the contracts found in the unsigned CorDapp JARs (a JAR file not signed by `jarSigner` tool). Whitelisted contracts are checked by *Zone constraints*, while contract classes from signed JARs will be checked by *Signature constraints*. To prevent certain contracts from unsigned JARs from being whitelisted, add their fully qualified class name in the `exclude_whitelist.txt`. These will instead use the more restrictive `HashAttachmentConstraint`. To add certain contracts from signed JARs to whitelist, add their fully qualified

class name in the `include_whitelist.txt`. Refer to [API: Contract Constraints](#) to understand the implication of different constraint types before adding `exclude_whitelist.txt` or `include_whitelist.txt` files.

For example:

```
net.corda.finance.contracts.asset.Cash
net.corda.finance.contracts.asset.CommercialPaper
```

## 10.2.4 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 Map](#) server.

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-4.1.jar --dir <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 Network 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 Network 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
```

(continues on next page)

(continued from previous page)

```

    └── 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-4.1.jar --dir <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.

## 10.2.5 Modifying the network parameters

The Network Bootstrapper creates a network parameters file when bootstrapping a network, using a set of sensible defaults. However, if you would like to override these defaults when testing, there are two ways of doing this. Options can be overridden via the command line or by supplying a configuration file. If the same parameter is overridden both by a command line argument and in the configuration file, the command line value will take precedence.

### Overriding network parameters via command line

The `--minimum-platform-version`, `--max-message-size`, `--max-transaction-size` and `--event-horizon` command line parameters can be used to override the default network parameters. See [Com-](#)

*mand line options* for more information.

## Overriding network parameters via a file

You can provide a network parameters overrides file using the following syntax:

```
java -jar network-bootstrapper-4.1.jar --network-parameter-overrides=<path_to_file>
```

Or alternatively, by using the short form version:

```
java -jar network-bootstrapper-4.1.jar -n=<path_to_file>
```

The network parameter overrides file is a HOCON file with the following fields, all of which are optional. Any field that is not provided will be ignored. If a field is not provided and you are bootstrapping a new network, a sensible default value will be used. If a field is not provided and you are updating an existing network, the value in the existing network parameters file will be used.

---

**Note:** All fields can be used with placeholders for environment variables. For example: \${KEY\_STORE\_PASSWORD} would be replaced by the contents of environment variable KEY\_STORE\_PASSWORD. See: [corda-configuration-hiding-sensitive-data](#).

---

The available configuration fields are listed below:

**minimumPlatformVersion** The minimum supported version of the Corda platform that is required for nodes in the network.

**maxMessageSize** The maximum permitted message size, in bytes. This is currently ignored but will be used in a future release.

**maxTransactionSize** The maximum permitted transaction size, in bytes.

**eventHorizon** The time after which nodes will be removed from the network map if they have not been seen during this period. This parameter uses the parse function on the java.time.Duration class to interpret the data. See [here](#) for information on valid inputs.

**packageOwnership** A list of package owners. See [Package namespace ownership](#) for more information. For each package owner, the following fields are required:

**packageName** Java package name (e.g `com.my_company` ).

**keystore** The path of the keystore file containing the signed certificate.

**keystorePassword** The password for the given keystore (not to be confused with the key password).

**keystoreAlias** The alias for the name associated with the certificate to be associated with the package namespace.

An example configuration file:

```
minimumPlatformVersion=4
maxMessageSize=10485760
maxTransactionSize=524288000
eventHorizon="30 days"
packageOwnership=[

{
    packageName="com.example"
    keystore="myteststore"
    keystorePassword="MyStorePassword"
```

(continues on next page)

(continued from previous page)

```

        keystoreAlias="MyKeyAlias"
    }
]
```

## 10.2.6 Package namespace ownership

Package namespace ownership is a Corda security feature that allows a compatibility zone to give ownership of parts of the Java package namespace to registered users (e.g. a CorDapp development organisation). The exact mechanism used to claim a namespace is up to the zone operator. A typical approach would be to accept an SSL certificate with the domain in it as proof of domain ownership, or to accept an email from that domain.

---

**Note:** Read more about *Package ownership* here.

---

A Java package namespace is case insensitive and cannot be a sub-package of an existing registered namespace. See [Naming a Package](#) and [Naming Conventions](#) for guidelines on naming conventions.

The registration of a Java package namespace requires the creation of a signed certificate as generated by the [Java keytool](#).

The packages can be registered by supplying a network parameters override config file via the command line, using the `--network-parameter-overrides` command.

For each package to be registered, the following are required:

**packageName** Java package name (e.g `com.my_company`).

**keystore** The path of the keystore file containing the signed certificate. If a relative path is provided, it is assumed to be relative to the location of the configuration file.

**keystorePassword** The password for the given keystore (not to be confused with the key password).

**keystoreAlias** The alias for the name associated with the certificate to be associated with the package namespace.

Using the [Example CorDapp](#) as an example, we will initialise a simple network and then register and unregister a package namespace. Checkout the Example CorDapp and follow the instructions to build it [here](#).

---

**Note:** You can point to any existing bootstrapped corda network (this will have the effect of updating the associated network parameters file).

---

1. Create a new public key to use for signing the Java package namespace we wish to register:

```
$JAVA_HOME/bin/keytool -genkeypair -keystore _teststore -storepass_
→MyStorePassword -keyalg RSA -alias MyKeyAlias -keypass MyKeyPassword -
→dname "O=Alice Corp, L=Madrid, C=ES"
```

This will generate a key store file called `_teststore` in the current directory.

2. Create a `network-parameters.conf` file in the same directory, with the following information:

```

packageOwnership=[
{
    packageName="com.example"
    keystore="_teststore"
}
```

(continues on next page)

(continued from previous page)

```

        keystorePassword="MyStorePassword"
        keystoreAlias="MyKeyAlias"
    }
]

```

3. Register the package namespace to be claimed by the public key generated above:

```

# Register the Java package namespace using the Network Bootstrapper
java -jar network-bootstrapper.jar --dir build/nodes --network-parameter-
→overrides=network-parameters.conf

```

4. To unregister the package namespace, edit the `network-parameters.conf` file to remove the package:

```
packageOwnership=[]
```

5. Unregister the package namespace:

```

# Unregister the Java package namespace using the Network Bootstrapper
java -jar network-bootstrapper.jar --dir build/nodes --network-parameter-
→overrides=network-parameters.conf

```

## 10.2.7 Command line options

The Network Bootstrapper can be started with the following command line options:

```
bootstrapper [-hvV] [--copy-cordapps=<copyCordapps>] [--dir=<dir>]
              [--event-horizon=<eventHorizon>] [--logging-level=<loggingLevel>]
              [--max-message-size=<maxMessageSize>]
              [--max-transaction-size=<maxTransactionSize>]
              [--minimum-platform-version=<minimumPlatformVersion>]
              [-n=<networkParametersFile>] [COMMAND]
```

- `--dir=<dir>`: Root directory containing the node configuration files and CorDapp JARs that will form the test network. It may also contain existing node directories. Defaults to the current directory.
- `--copy-cordapps=<copyCordapps>`: Whether or not to copy the CorDapp JARs into the nodes' 'cordapps' directory. Possible values: FirstRunOnly, Yes, No. Default: FirstRunOnly.
- `--verbose, --log-to-console, -v`: If set, prints logging to the console as well as to a file.
- `--logging-level=<loggingLevel>`: Enable logging at this level and higher. Possible values: ERROR, WARN, INFO, DEBUG, TRACE. Default: INFO.
- `--help, -h`: Show this help message and exit.
- `--version, -V`: Print version information and exit.
- `--minimum-platform-version`: The minimum platform version to use in the `network-parameters`.
- `--max-message-size`: The maximum message size to use in the `network-parameters`, in bytes.
- `--max-transaction-size`: The maximum transaction size to use in the `network-parameters`, in bytes.
- `--event-horizon`: The event horizon to use in the `network-parameters`.
- `--network-parameter-overrides=<networkParametersFile>, -n=<networkParametersFile>`: Overrides the default network parameters with those in the given file. See [Overriding network parameters via a file](#) for more information.

## Sub-commands

`install-shell-extensions`: Install bootstrapper alias and auto completion for bash and zsh. See [Shell extensions for CLI Applications](#) for more info.

## 10.3 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.

### 10.3.1 Downloading

Installers compatible with the latest Corda release can be downloaded from the [Corda website](#).

### 10.3.2 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	\$HOME/demobench/demobench.log
Windows	%USERPROFILE%\demobench\demobench.log

### 10.3.3 Building the Installers

Gradle defines tasks that build DemoBench installers using JavaPackager. There are three scripts in the `tools/demobench` directory of the [Corda repository](#) 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  $\geq 8u152$  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.

### 10.3.4 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.

### 10.3.5 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
```

## 10.4 Node Explorer

---

**Note:** To run Node Explorer on your machine, you will need JavaFX for Java 8. If you don't have JavaFX installed, you can either download and build your own version of OpenJFK, or use a pre-existing build, like the one offered by Zulu. They have community builds of OpenJFX for Window, macOS and Linux available on their [website](#).

---

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)

### 10.4.1 Running the UI

**Windows:**

```
gradlew.bat tools:explorer:run
```

**Other:**

```
./gradlew tools:explorer:run
```

---

**Note:** In order to connect to a given node, the node explorer must have access to all CorDapps loaded on that particular node. By default, it only has access to the finance CorDapp. All other CorDapps present on the node must be copied to a `cordapps` directory located within the directory from which the node explorer is run.

---

## 10.4.2 Running demo nodes

Node Explorer is included with the *DemoBench* application, which allows you to create local Corda networks on your desktop. For example:

- Notary
- Bank of Breakfast Tea (*Issuer node* for GBP)
- Bank of Big Apples (*Issuer node* for USD)
- Alice (*Participant node*, for user Alice)
- Bob (*Participant node*, for user Bob)

DemoBench will deploy all nodes with Corda's Finance CorDapp automatically, and allow you to launch an instance of Node Explorer for each. You will also be logged into the Node Explorer automatically.

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 Node Explorer is also available as a stand-alone JavaFX application. It is available from the Corda repositories as `corda-tools-explorer`, and can be run as

```
java -jar corda-tools-explorer.jar
```

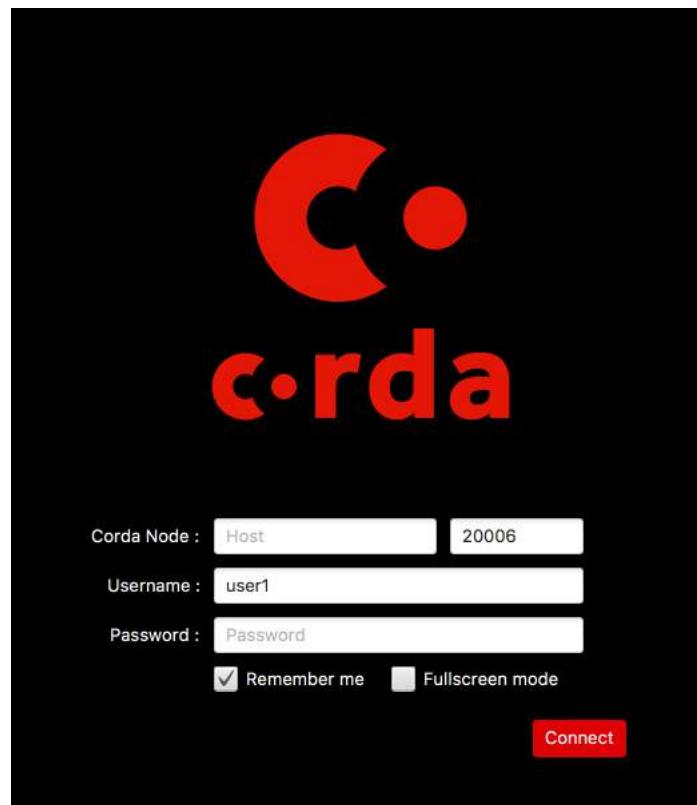
---

**Note:** Use the Explorer in conjunction with the Trader Demo and Bank of Corda samples to use other *Issuer* nodes.

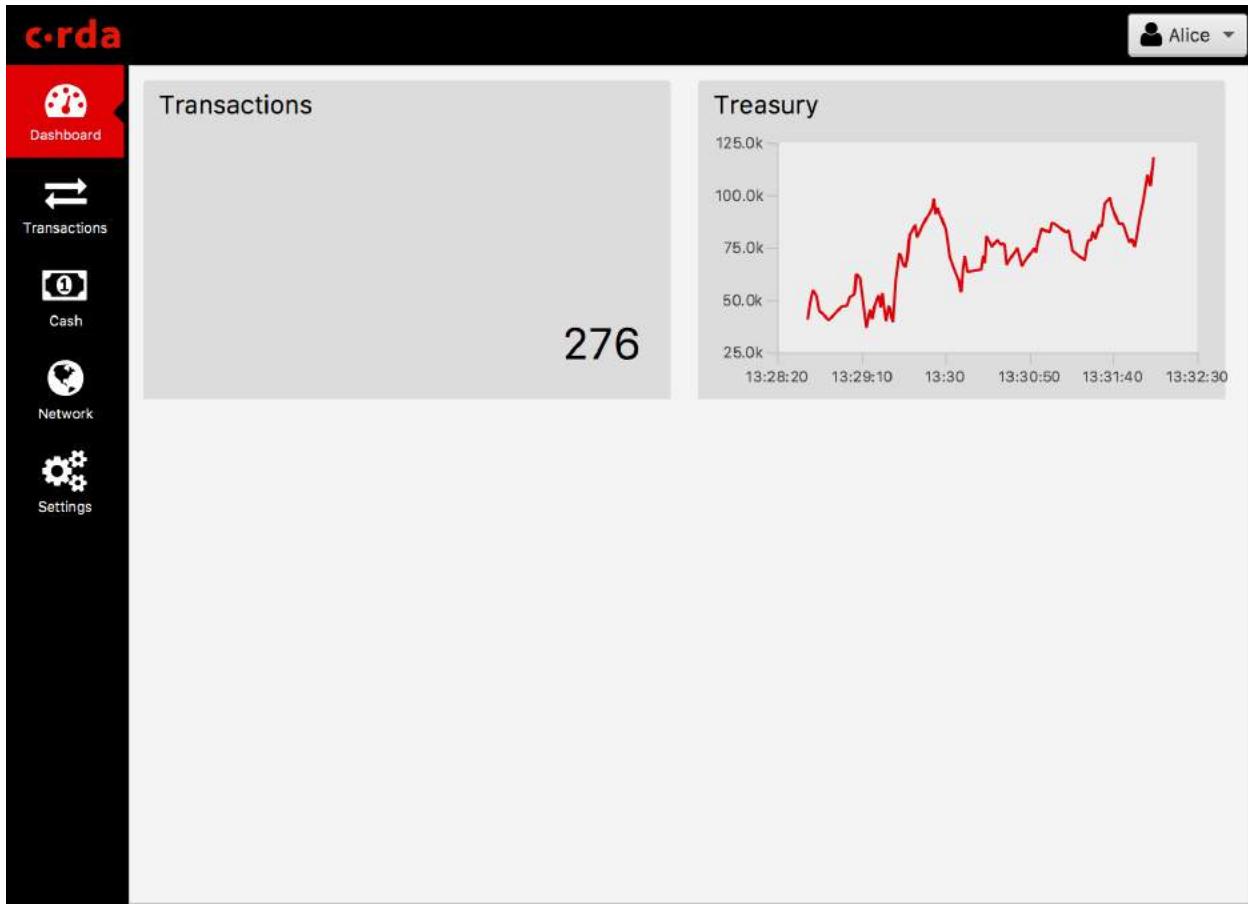
---

## 10.4.3 Interface

**Login** User can login to any Corda node using the explorer. Corda node address, username and password are required for login, the address is defaulted to localhost:0 if left 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 Corda Node Explorer interface. On the left is a sidebar with icons for Dashboard, Transactions, Cash (selected), 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'. The main table lists issuers and their cash positions:

Issuer/Currency	Local currency	USD Equiv
Federal Reserve	909287 USD	
GBP	291859 GBP	291859 USD
CHF	303551 CHF	303551 USD
USD	313877 USD	313877 USD
Royal Mint	18800 USD	
GBP	861 GBP	861 USD
CHF	7919 CHF	7919 USD
USD	10020 USD	10020 USD

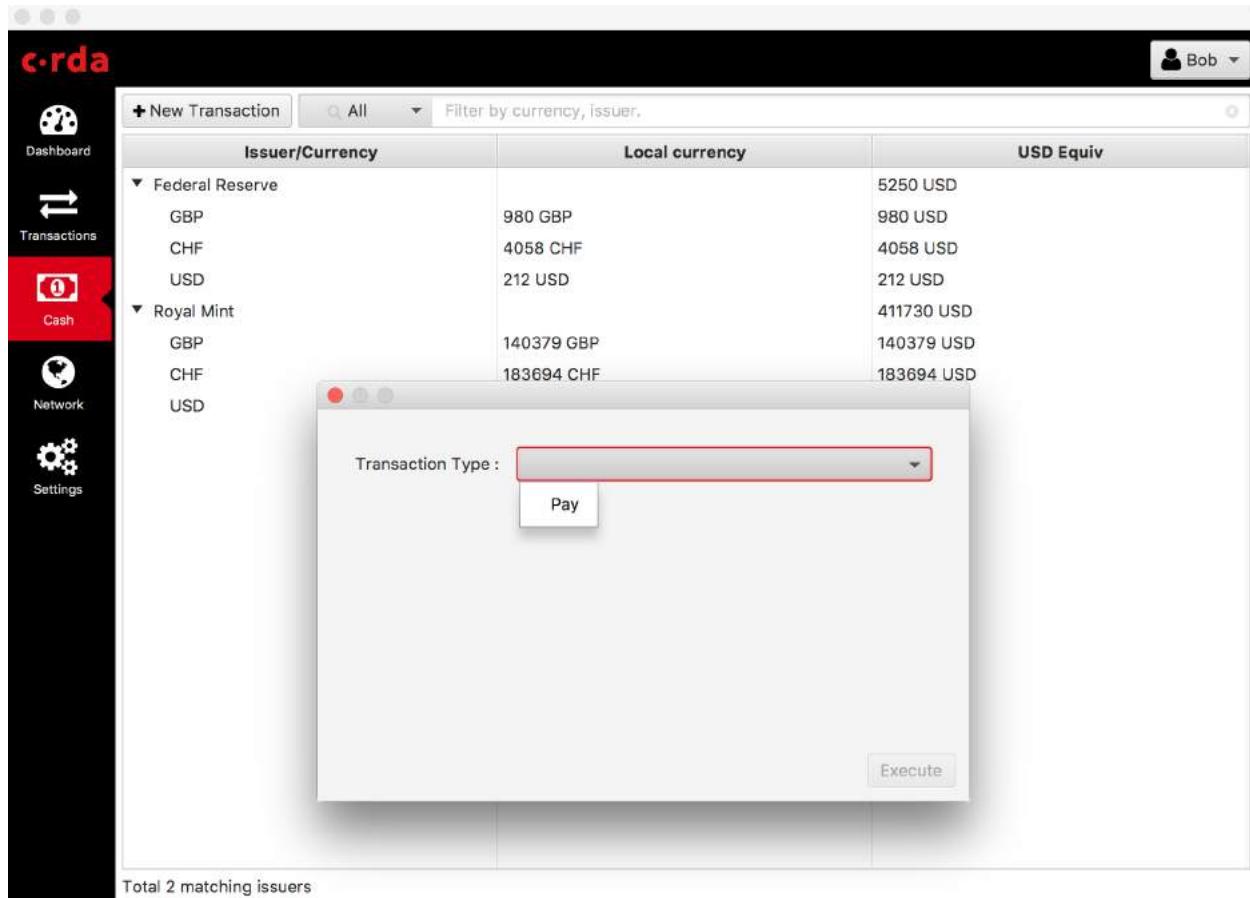
Below the table, it says 'Total 2 matching issuers'. To the right is a detailed view of the Federal Reserve's cash positions:

- 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]

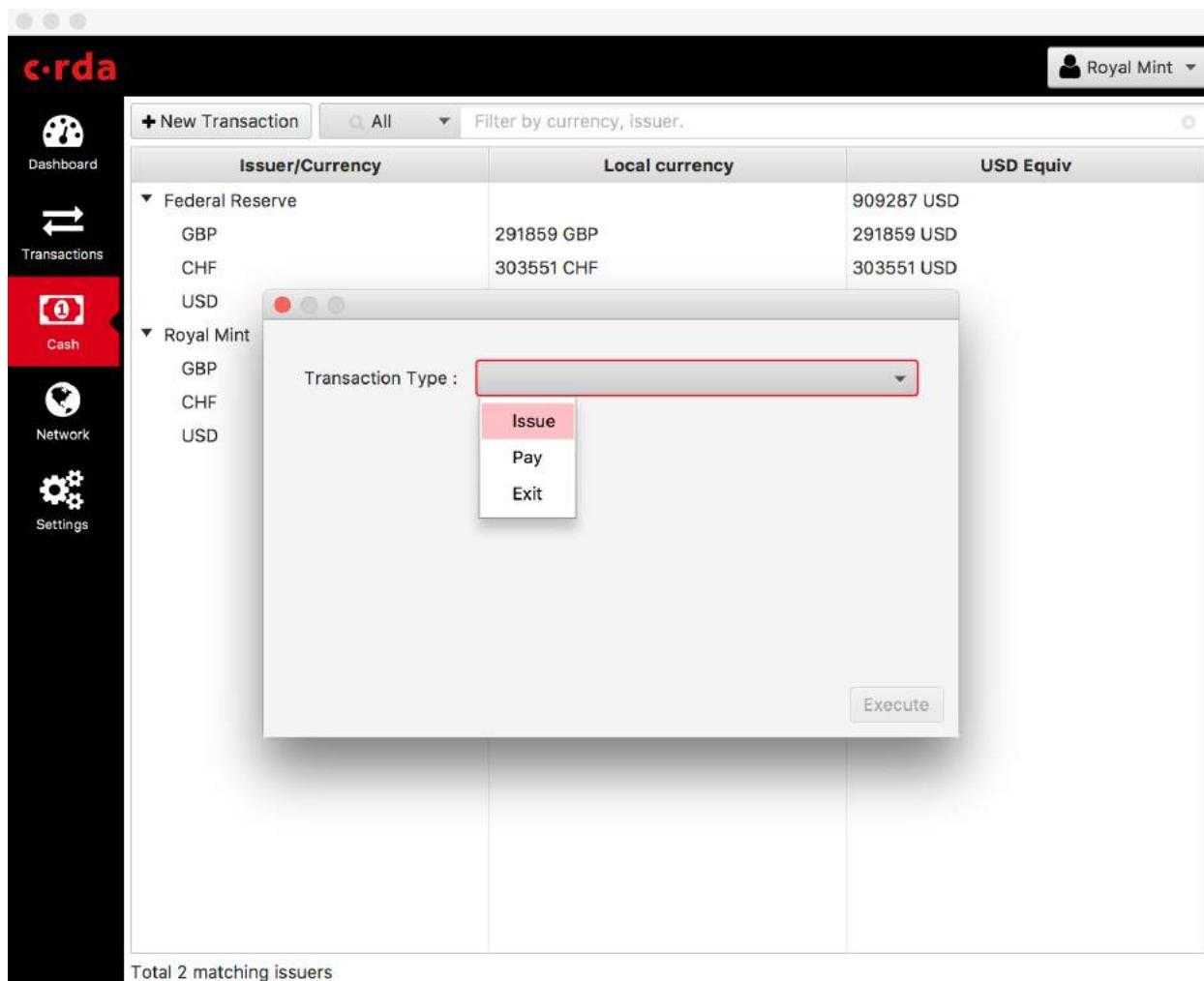
Total 94 positions

**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.

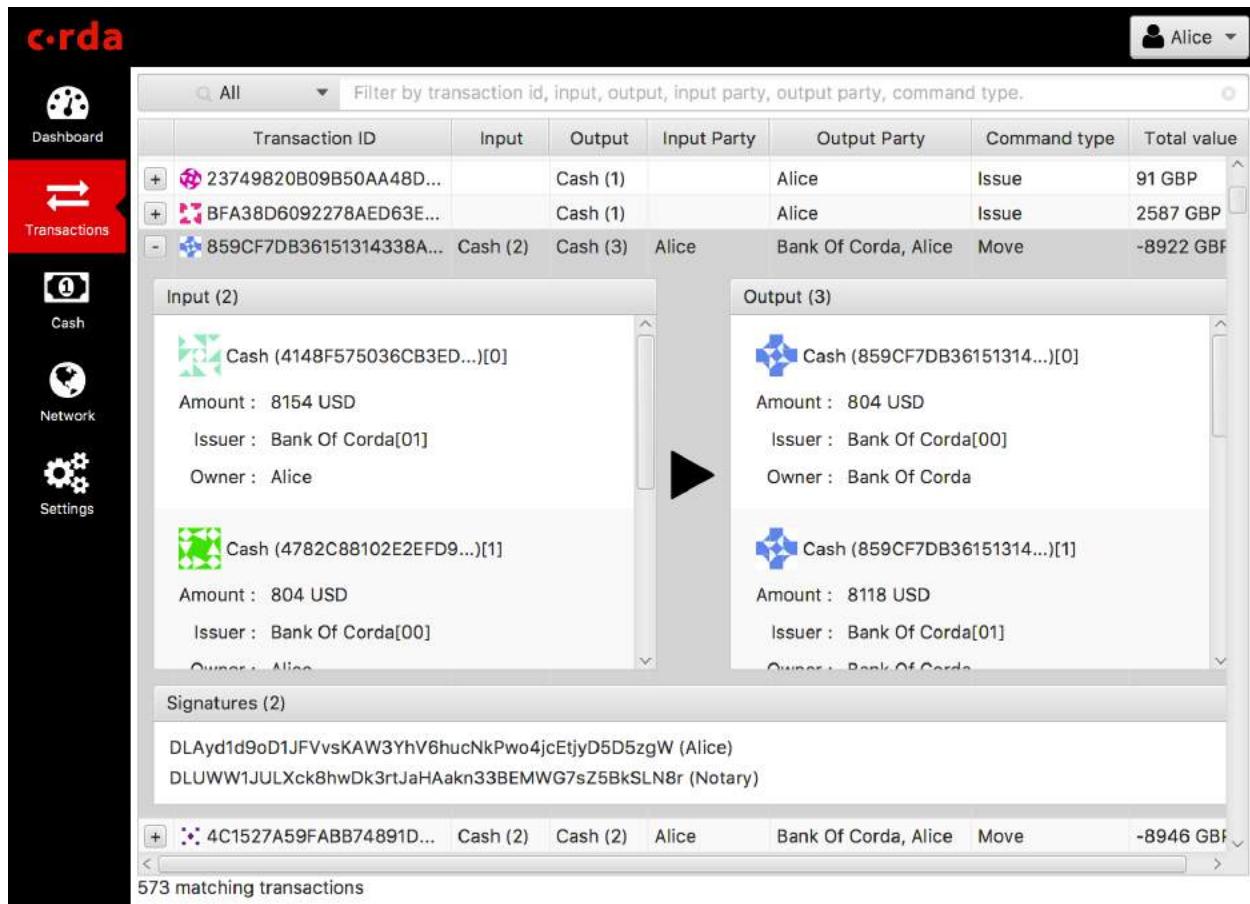
General nodes can only execute pay commands to any other party on the network.



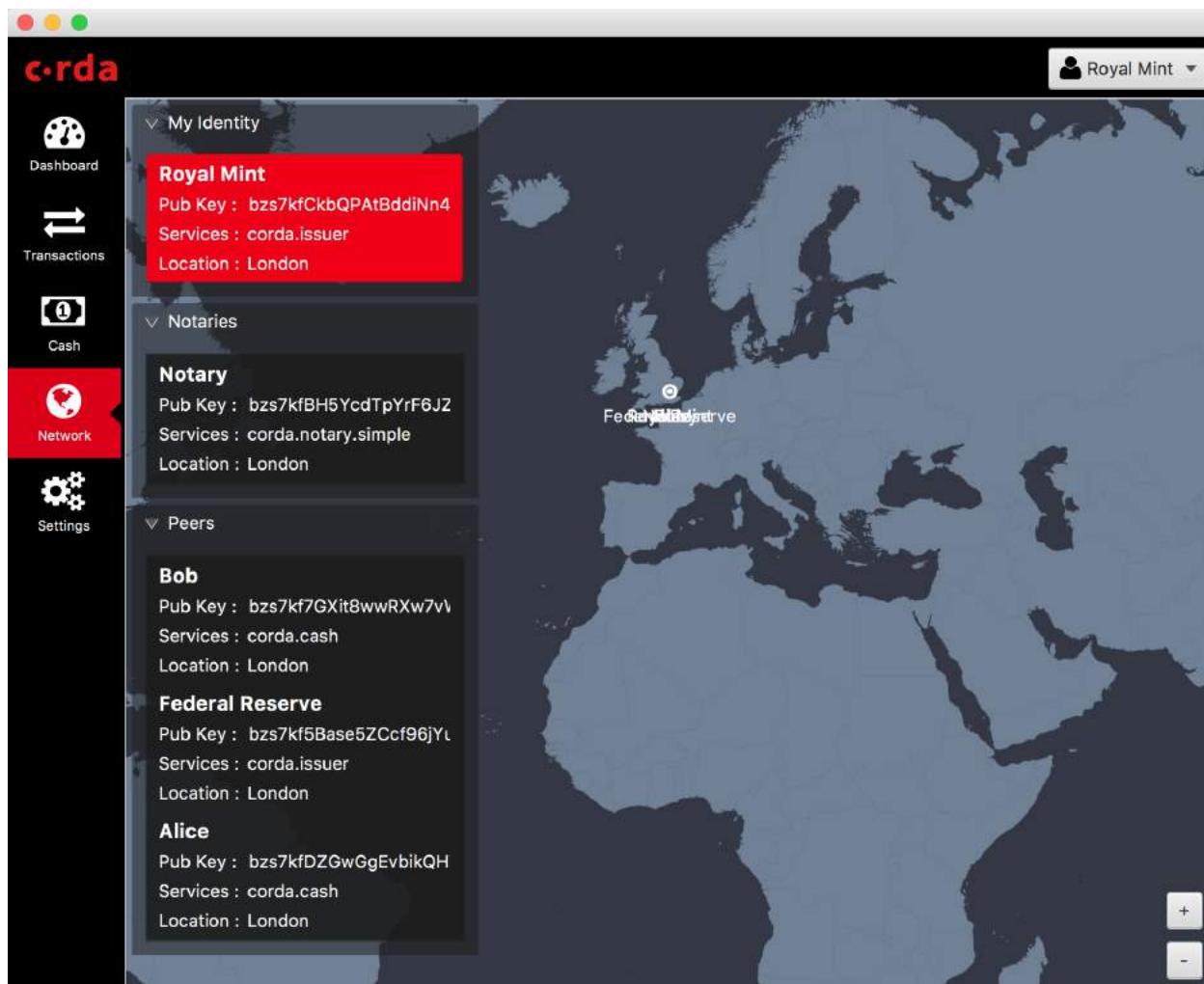
**Issuer Nodes** Issuer nodes can execute issue (to itself or to any other party), pay and exit transactions. The result of the transaction will be visible in the transaction screen when executed.



**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.



**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.

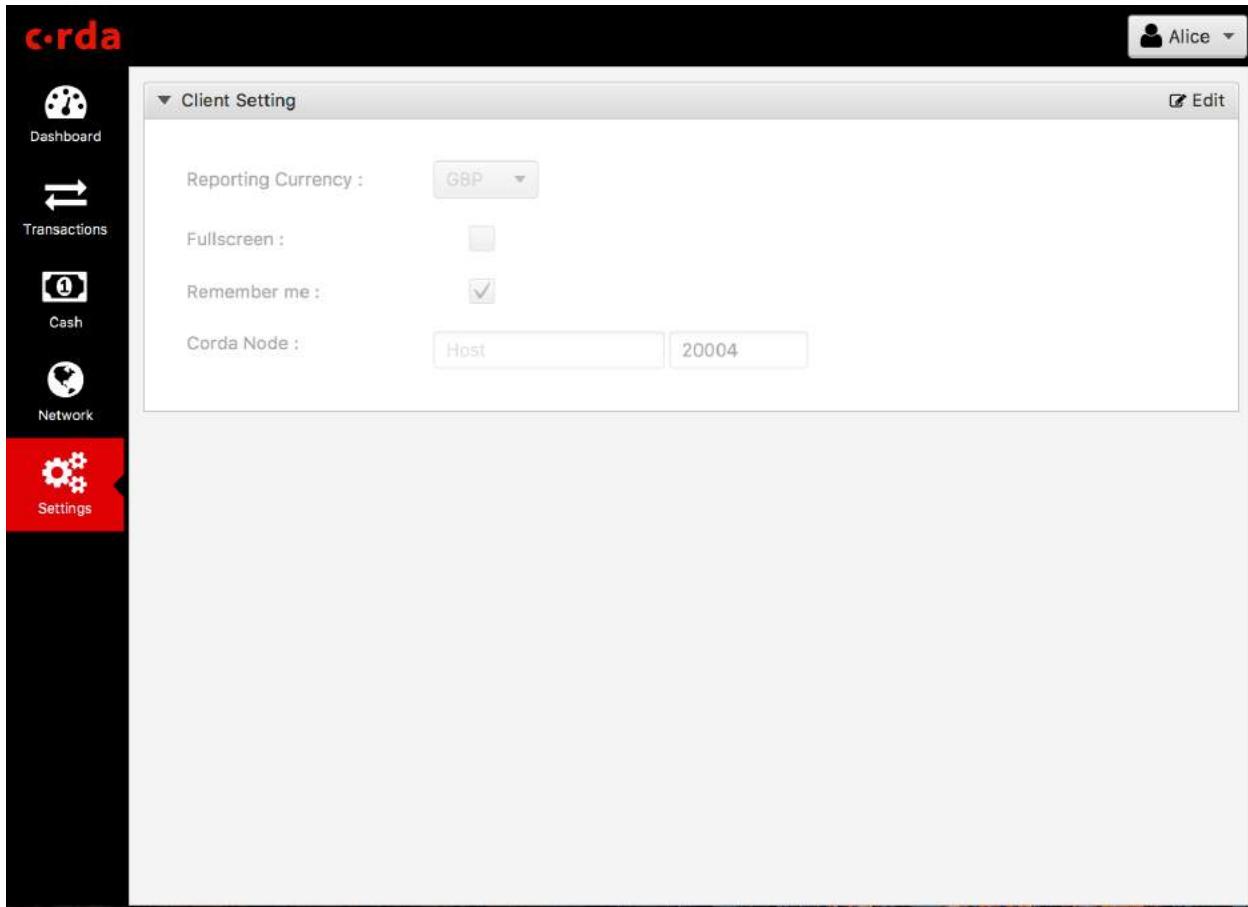


**Settings** User can configure the client preference in this view.

---

**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.

---



## NODE INTERNALS

### 11.1 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.

#### 11.1.1 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 `net.corda.core.node.services` package. The `ServiceHub` interface exposes functionality suitable for flows. The implementation code for all standard services lives in the `net.corda.node.services` package.

All the services are constructed in the `AbstractNode.start` method. 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.

The roles of the individual services are described below.

#### 11.1.2 Key management and identity services

##### **InMemoryIdentityService**

The `InMemoryIdentityService` implements the `IdentityService` interface and provides a store of remote mappings between `PublicKey` and remote `Parties`. It is automatically populated from the `NetworkMapCache` updates and is used when translating `PublicKey` 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 `PublicKey` 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.

### 11.1.3 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 `NetworkMapCache`. 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 `NetworkMapCache` and thus must be externally connectable by all nodes in the network.

#### P2PMessagingClient

The `P2PMessagingClient` 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 `PublicKey`, or `Party` based addressing used in the flows/contracts and the physical host and port information required for the physical `ArtemisMQ` messaging layer.

### 11.1.4 Storage and persistence related services

## 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.

### 11.1.5 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.

## **11.1.6 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.

## **11.1.7 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.

---

## **11.2 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.

### 11.2.1 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 backed up in the database to persist that information across the restarts in case the network map server is down). When a node starts up its cache fetches a copy of the full network map (from the server or from filesystem for development mode). After that it polls on regular time interval for network map and applies any related changes locally. 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.

Additionally, on every restart and on daily basis nodes submit signed `NodeInfo`s to the map service. When network map gets signed, these changes are distributed as new network data. `NodeInfo` republishing is treated as a heartbeat from the node, based on that network map service is able to figure out which nodes can be considered as stale and removed from the network map document after `eventHorizon` time.

### 11.2.2 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.server`** RPC clients send their requests here, and it's only open for sending by clients authenticated as RPC users.

**rpc.client.\$user.\$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 to listen in on the responses.

### 11.2.3 Security

Clients attempting to connect to the node's broker fall in one of four groups:

1. Anyone connecting with the username SystemUsers/Node or SystemUsers/NodeRPC is treated as the node hosting the brokers, 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.

### Implementation details

**The components of the system that need to communicate and authenticate each other are:**

- **The Artemis P2P broker (currently runs inside the node's JVM process, but in the future it will be able to run as a separate process):**
  - Opens Acceptor configured with the doorman's certificate in the trustStore and the node's SSL certificate in the keyStore.
- **The Artemis RPC broker (currently runs inside the node's JVM process, but in the future it will be able to run as a separate process):**
  - Opens “Admin” Acceptor configured with the doorman's certificate in the trustStore and the node's SSL certificate in the keyStore.
  - Opens “Client” Acceptor with the SSL settings configurable. This acceptor does not require SSL client-auth.
- **The current node hosting the brokers:**
  - Connects to the P2P broker using the SystemUsers/Node user and the node's keyStore and trustStore.

- Connects to the “Admin” Acceptor of the RPC broker using the `SystemUsers/NodeRPC` user and the node’s keyStore and trustStore.
- **RPC clients (third party applications that need to communicate with the node):**
  - Connect to the “Client” Acceptor of the RPC broker using the username/password provided by the node’s admin. The client verifies the node’s certificate using a trustStore provided by the node’s admin.
- **Peer nodes (other nodes on the network):**
  - Connect to the P2P broker using the `SystemUsers/Peer` user and a doorman signed certificate. The authentication is performed based on the root CA.

## COMPONENT LIBRARY

### 12.1 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.

#### 12.1.1 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`

---

#### 12.1.2 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.

#### 12.1.3 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.

### 12.1.4 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.

### 12.1.5 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).

## 12.2 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.

### 12.2.1 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 are 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.

## 12.2.2 Financial states

In addition to the common state types, a number of interfaces extend `ContractState` to model financial state such as:

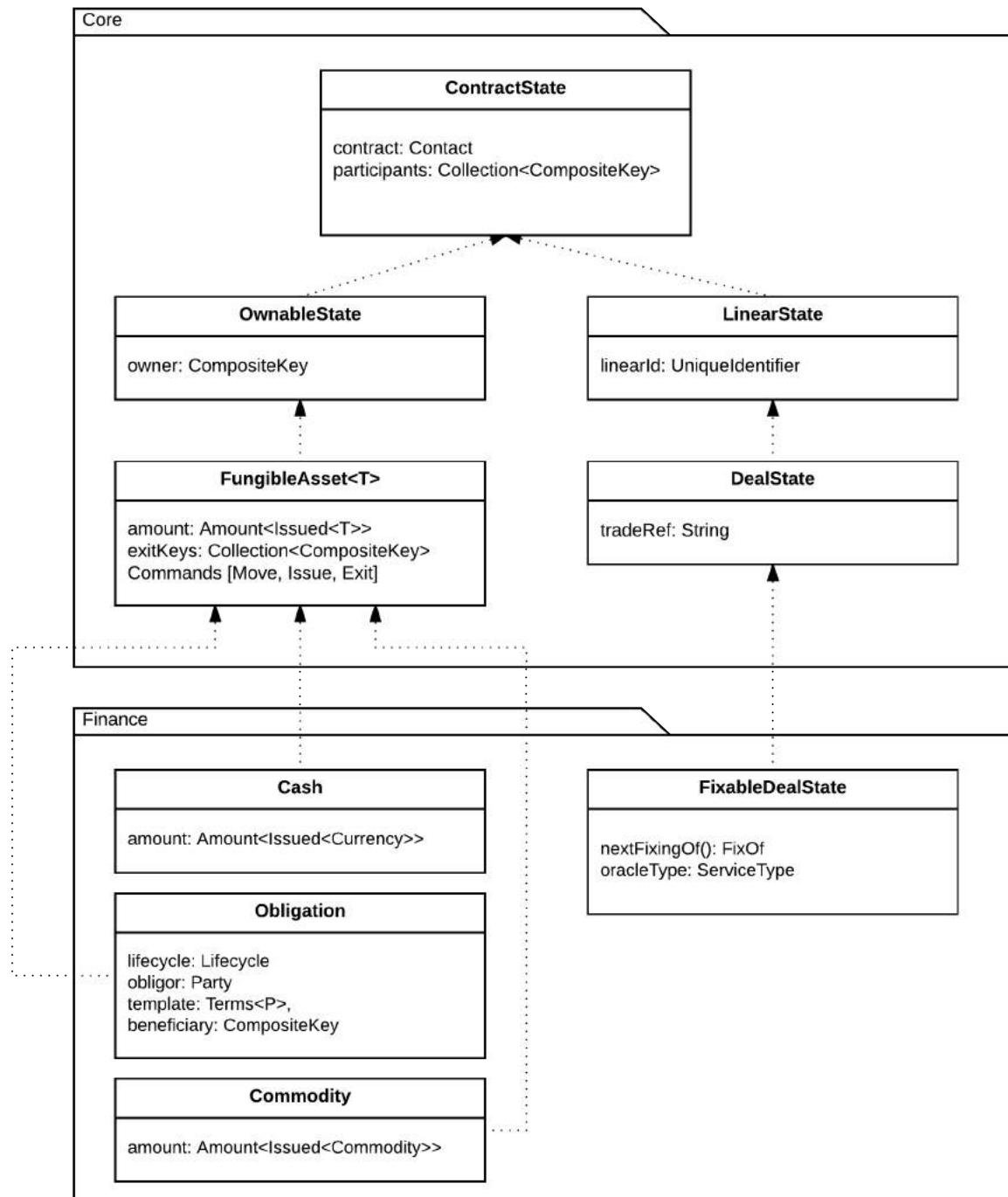
**LinearState** A state which has a unique identifier beyond its `StateRef` and carries it through state transitions. Such a state cannot be duplicated, merged or split in a transaction: only continued or deleted. A linear state is useful when modelling an indivisible/non-fungible thing like a specific 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.

## 12.3 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.

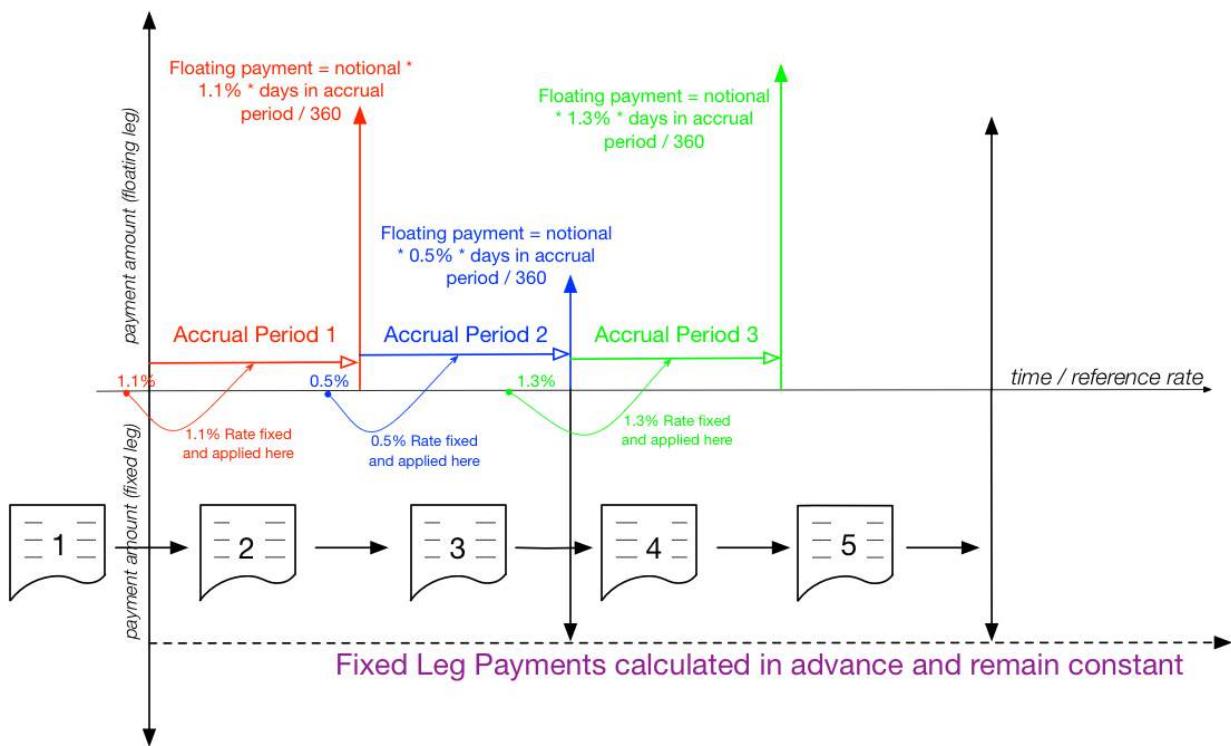
The IRS contract exists over a period of time (normally measurable in years). It starts on its value date (although this is not the agreement date), and is considered to be no longer active after its maturity date. During that time, there is an exchange of cash flows which are calculated by looking at 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.

### 12.3.1 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.

### 12.3.2 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`).

---

## SERIALIZATION

### 13.1 Object serialization

#### Contents

- *Object serialization*
  - *Introduction*
  - *Whitelisting*
  - *AMQP*
  - *Core Types*
    - \* *Collection Types*
    - \* *JVM primitives*
    - \* *Arrays*
    - \* *JDK Types*
    - \* *Third-Party Types*
    - \* *Corda Types*
  - *Custom Types*
    - \* *Classes*
      - *General Rules*
      - *Constructor Instantiation*
      - *Setter Instantiation*
    - \* *Inaccessible Private Properties*
    - \* *Mismatched Class Properties / Constructor Parameters*
    - \* *Mutable Containers*
    - \* *Enums*
    - \* *Exceptions*
    - \* *Kotlin Objects*
  - *Class synthesis*

- \* *Calculated values*
- \* *Future enhancements*
- *Type Evolution*

### 13.1.1 Introduction

Object serialization is the process of converting objects into a stream of bytes and, deserialization, the reverse process of creating objects from a stream of bytes. It takes place every time nodes pass objects to each other as messages, when objects are sent to or from RPC clients from the node, and when we store transactions in the database.

Corda pervasively uses a custom form of type safe binary serialisation. This stands in contrast to some other systems that use weakly or untyped string-based serialisation schemes like JSON or XML. The primary drivers for this were:

- A desire to have a schema describing what has been serialized alongside the actual data:
  1. To assist with versioning, both in terms of being able to interpret data archived long ago (e.g. trades from a decade ago, long after the code has changed) and between differing code versions.
  2. To make it easier to write generic code e.g. user interfaces that can navigate the serialized form of data.
  3. To support cross platform (non-JVM) interaction, where the format of a class file is not so easily interpreted.
- A desire to use a documented and static wire format that is platform independent, and is not subject to change with 3rd party library upgrades, etc.
- A desire to support open-ended polymorphism, where the number of subclasses of a superclass can expand over time and the subclasses do not need to be defined in the schema *upfront*. This is key to many Corda concepts, such as states.
- Increased security by constructing deserialized objects through supported constructors, rather than having data inserted directly into their fields without an opportunity to validate consistency or intercept attempts to manipulate supposed invariants.
- Binary formats work better with digital signatures than text based formats, as there's much less scope for changes that modify syntax but not semantics.

### 13.1.2 Whitelisting

In classic Java serialization, any class on the JVM classpath can be deserialized. This is a source of exploits and vulnerabilities that exploit the large set of third-party libraries that are added to the classpath as part of a JVM application's dependencies and carefully craft a malicious stream of bytes to be deserialized. In Corda, we strictly control which classes can be serialized (and, proactively, serialized) by insisting that each (de)serializable class is part of a whitelist of allowed classes.

To add a class to the whitelist, you must use either of the following mechanisms:

1. Add the `@CordaSerializable` annotation to the class. This annotation can be present on the class itself, on any super class of the class, on any interface implemented by the class or its super classes, or any interface extended by an interface implemented by the class or its super classes.
2. Implement the `SerializationWhitelist` interface and specify a list of whitelisted classes.

There is also a built-in Corda whitelist (see the `DefaultWhitelist` class) that whitelists common JDK classes for convenience. This whitelist is not user-editable.

The annotation is the preferred method for whitelisting. An example is shown in [Using the client RPC API](#). It's reproduced here as 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)
}
```

---

**Note:** Several of the core interfaces at the heart of Corda are already annotated and so any classes that implement them will automatically be whitelisted. This includes `Contract`, `ContractState` and `CommandData`.

---

**Warning:** Java 8 Lambda expressions are not serializable except in flow checkpoints, and then not by default. The syntax to declare a serializable Lambda expression that will work with Corda is `Runnable r = (Runnable & Serializable) () -> System.out.println("Hello World");`, or `Callable<String> c = (Callable<String> & Serializable) () -> "Hello World";`.

### 13.1.3 AMQP

Corda uses an extended form of AMQP 1.0 as its binary wire protocol.

Corda serialisation is currently used for:

1. Peer-to-peer networking.
2. Persisted messages, like signed transactions and states.

For the checkpointing of flows Corda uses a private scheme that is subject to change. It is currently based on the Kryo framework, but this may not be true in future.

This separation of serialization schemes into different contexts allows us to use the most suitable framework for that context rather than attempting to force a one-size-fits-all approach. Kryo is more suited to the serialization of a program's stack frames, as it is more flexible than our AMQP framework in what it can construct and serialize. However, that flexibility makes it exceptionally difficult to make secure. Conversely, our AMQP framework allows us to concentrate on a secure framework that can be reasoned about and thus made safer, with far fewer security holes.

Selection of serialization context should, for the most part, be opaque to CorDapp developers, the Corda framework selecting the correct context as configured.

This document describes what is currently and what will be supported in the Corda AMQP format from the perspective of CorDapp developers, to allow CorDapps to take into consideration the future state. The AMQP serialization format will continue to apply the whitelisting functionality that is already in place and described in [Object serialization](#).

### 13.1.4 Core Types

This section describes the classes and interfaces that the AMQP serialization format supports.

## Collection Types

The following collection types are supported. Any implementation of the following will be mapped to *an* implementation of the interface or class on the other end. For example, if you use a Guava implementation of a collection, it will deserialize as the primitive collection type.

The declared types of properties should only use these types, and not any concrete implementation types (e.g. Guava implementations). Collections must specify their generic type, the generic type parameters will be included in the schema, and the element's type will be checked against the generic parameters when deserialized.

```
java.util.Collection  
java.util.List  
java.util.Set  
java.util.SortedSet  
java.util.NavigableSet  
java.util.NonEmptySet  
java.util.Map  
java.util.SortedMap  
java.util.NavigableMap
```

However, as a convenience, we explicitly support the concrete implementation types below, and they can be used as the declared types of properties.

```
java.util.LinkedHashMap  
java.util.TreeMap  
java.util.EnumSet  
java.util.EnumMap (but only if there is at least one entry)
```

## JVM primitives

All the primitive types are supported.

```
boolean  
byte  
char  
double  
float  
int  
long  
short
```

## Arrays

Arrays of any type are supported, primitive or otherwise.

## JDK Types

The following JDK library types are supported:

```
java.io.InputStream  
  
java.lang.Boolean  
java.lang.Byte
```

(continues on next page)

(continued from previous page)

```
java.lang.Character  
java.lang.Class  
java.lang.Double  
java.lang.Float  
java.lang.Integer  
java.lang.Long  
java.lang.Short  
java.lang.StackTraceElement  
java.lang.String  
java.lang.StringBuffer  
  
java.math.BigDecimal  
  
java.security.PublicKey  
  
java.time.DayOfWeek  
java.time.Duration  
java.time.Instant  
java.time.LocalDate  
java.time.LocalDateTime  
java.time.LocalTime  
java.time.Month  
java.time.MonthDay  
java.time.OffsetDateTime  
java.time.OffsetTime  
java.time.Period  
java.time.YearMonth  
java.time.Year  
java.time.ZonedDateTime  
java.time.ZonedId  
java.time.ZoneOffset  
  
java.util.BitSet  
java.util.Currency  
java.util.UUID
```

## Third-Party Types

The following 3rd-party types are supported:

```
kotlin.Unit  
kotlin.Pair  
  
org.apache.activemq.artemis.api.core.SimpleString
```

## Corda Types

Any classes and interfaces in the Corda codebase annotated with @CordaSerializable are supported.

All Corda exceptions that are expected to be serialized inherit from CordaThrowable via either CordaException (for checked exceptions) or CordaRuntimeException (for unchecked exceptions). Any Throwable that is serialized but does not conform to CordaThrowable will be converted to a CordaRuntimeException, with the original exception type and other properties retained within it.

### 13.1.5 Custom Types

You own types must adhere to the following rules to be supported:

#### Classes

#### General Rules

1. The class must be compiled with parameter names included in the `.class` file. This is the default in Kotlin but must be turned on in Java using the `-parameters` command line option to `javac`

---

**Note:** In circumstances where classes cannot be recompiled, such as when using a third-party library, a proxy serializer can be used to avoid this problem. Details on creating such an object can be found on the [Pluggable Serializers for CorDapps](#) page.

---

2. The class must be annotated with `@CordaSerializable`
3. The declared types of constructor arguments, getters, and setters must be supported, and where generics are used, the generic parameter must be a supported type, an open wildcard `(*)`, or a bounded wildcard which is currently widened to an open wildcard
4. Any superclass must adhere to the same rules, but can be abstract
5. Object graph cycles are not supported, so an object cannot refer to itself, directly or indirectly

#### Constructor Instantiation

The primary way Corda's AMQP serialization framework instantiates objects is via a specified constructor. This is used to first determine which properties of an object are to be serialised, then, on deserialization, it is used to instantiate the object with the serialized values.

It is recommended that serializable objects in Corda adhere to the following rules, as they allow immutable state objects to be deserialised:

1. A Java Bean getter for each of the properties in the constructor, with a name of the form `getX`. For example, for a constructor parameter `foo`, there must be a getter called `getFoo()`. If `foo` is a boolean, the getter may optionally be called `isFoo()` (this is why the class must be compiled with parameter names turned on)
2. A constructor which takes all of the properties that you wish to record in the serialized form. This is required in order for the serialization framework to reconstruct an instance of your class
3. If more than one constructor is provided, the serialization framework needs to know which one to use. The `@ConstructorForDeserialization` annotation can be used to indicate which one. For a Kotlin class, without the `@ConstructorForDeserialization` annotation, the *primary constructor* will be selected

In Kotlin, this maps cleanly to a data class where there getters are synthesized automatically. For example, suppose we have the following data class:

```
data class Example (val a: Int, val b: String)
```

Properties `a` and `b` will be included in the serialised form.

However, properties not mentioned in the constructor will not be serialised. For example, in the following code, property `c` will not be considered part of the serialised form:

```

data class Example (val a: Int, val b: String) {
    var c: Int = 20
}

var e = Example (10, "hello")
e.c = 100;

val e2 = e.serialize().deserialize() // e2.c will be 20, not 100!!!

```

## Setter Instantiation

As an alternative to constructor-based initialisation, Corda can also determine the important elements of an object by inspecting the getter and setter methods present on the class. If a class has **only** a default constructor **and** properties then the serializable properties will be determined by the presence of both a getter and setter for that property that are both publicly visible (i.e. the class adheres to the classic *idiotm* of mutable JavaBeans).

On deserialization, a default instance will first be created, and then the setters will be invoked on that object to populate it with the correct values.

For example:

```
class Example(var a: Int, var b: Int, var c: Int)
```

```

class Example {
    private int a;
    private int b;
    private int c;

    public int getA() { return a; }
    public int getB() { return b; }
    public int getC() { return c; }

    public void setA(int a) { this.a = a; }
    public void setB(int b) { this.b = b; }
    public void setC(int c) { this.c = c; }
}

```

**Warning:** We do not recommend this pattern! Corda tries to use immutable data structures throughout, and if you rely heavily on mutable JavaBean style objects then you may sometimes find the API behaves in unintuitive ways.

## Inaccessible Private Properties

Whilst the Corda AMQP serialization framework supports private object properties without publicly accessible getter methods, this development idiom is strongly discouraged.

For example,

```
class C(val a: Int, private val b: Int)
```

```

class C {
    public Integer a;
    private Integer b;
}

```

(continues on next page)

(continued from previous page)

```
public C(Integer a, Integer b) {
    this.a = a;
    this.b = b;
}
```

When designing Corda states, it should be remembered that they are not, despite appearances, traditional OOP style objects. They are signed over, transformed, serialised, and relationally mapped. As such, all elements should be publicly accessible by design.

**Warning:** IDEs will indicate erroneously that properties can be given something other than public visibility. Ignore this, as whilst it will work, as discussed above there are many reasons why this isn't a good idea.

Providing a public getter, as per the following example, is acceptable:

```
class C(val a: Int, b: Int) {
    var b: Int = b
        private set
}
```

```
class C {
    public Integer a;
    private Integer b;

    C(Integer a, Integer b) {
        this.a = a;
        this.b = b;
    }

    public Integer getB() {
        return b;
    }
}
```

## Mismatched Class Properties / Constructor Parameters

Consider an example where you wish to ensure that a property of class whose type is some form of container is always sorted using some specific criteria yet you wish to maintain the immutability of the class.

This could be codified as follows:

```
@CordaSerializable
class ConfirmRequest(statesToConsume: List<StateRef>, val transactionId: SecureHash) {
    companion object {
        private val stateRefComparator = compareBy<StateRef>({ it.txhash }, { it.
            index })
    }

    private val states = statesToConsume.sortedWith(stateRefComparator)
}
```

The intention in the example is to always ensure that the states are stored in a specific order regardless of the ordering of the list used to initialise instances of the class. This is achieved by using the first constructor parameter as the basis

for a private member. However, because that member is not mentioned in the constructor (whose parameters determine what is serializable as discussed above) it would not be serialized. In addition, as there is no provided mechanism to retrieve a value for `statesToConsume` we would fail to build a serializer for this Class.

In this case a secondary constructor annotated with `@ConstructorForDeserialization` would not be a valid solution as the two signatures would be the same. Best practice is thus to provide a getter for the constructor parameter which explicitly associates it with the actual member variable.

```
@CordaSerializable
class ConfirmRequest(statesToConsume: List<StateRef>, val transactionId: SecureHash) {
    companion object {
        private val stateRefComparator = compareBy<StateRef>({ it.txhash }, { it.
            index })
    }

    private val states = statesToConsume.sortedWith(stateRefComparator)

    //Explicit "getter" for a property identified from the constructor parameters
    fun getStatesToConsume() = states
}
```

## Mutable Containers

Because Java fundamentally provides no mechanism by which the mutability of a class can be determined this presents a problem for the serialization framework. When reconstituting objects with container properties (lists, maps, etc) we must chose whether to create mutable or immutable objects. Given the restrictions, we have decided it is better to preserve the immutability of immutable objects rather than force mutability on presumed immutable objects.

---

**Note:** Whilst we could potentially infer mutability empirically, doing so exhaustively is impossible as it's a design decision rather than something intrinsic to the JVM. At present, we defer to simply making things immutable on reconstruction with the following workarounds provided for those who use them. In future, this may change, but for now use the following examples as a guide.

For example, consider the following:

```
data class C(val l : MutableList<String>)

val bytes = C(mutableListOf ("a", "b", "c")).serialize()
val newC = bytes.deserialize()

newC.l.add("d")
```

The call to `newC.l.add` will throw an `UnsupportedOperationException`.

There are several workarounds that can be used to preserve mutability on reconstituted objects. Firstly, if the class isn't a Kotlin data class and thus isn't restricted by having to have a primary constructor.

```
class C {
    val l : MutableList<String>

    @Suppress("Unused")
    constructor (l : MutableList<String>) {
        this.l = l.toMutableList()
    }
}
```

(continues on next page)

(continued from previous page)

```
val bytes = C(mutableListOf ("a", "b", "c")).serialize()
val newC = bytes.deserialize()

// This time this call will succeed
newC.l.add("d")
```

Secondly, if the class is a Kotlin data class, a secondary constructor can be used.

```
data class C (val l : MutableList<String>) {
    @ConstructorForDeserialization
    @Suppress("Unused")
    constructor (l : Collection<String>) : this (l.toMutableList())
}

val bytes = C(mutableListOf ("a", "b", "c")).serialize()
val newC = bytes.deserialize()

// This will also work
newC.l.add("d")
```

Thirdly, to preserve immutability of objects (a recommend design principle - Copy on Write semantics) then mutating the contents of the class can be done by creating a new copy of the data class with the altered list passed (in this example) passed in as the Constructor parameter.

```
data class C(val l : List<String>

val bytes = C(listOf ("a", "b", "c")).serialize()
val newC = bytes.deserialize()

val newC2 = newC.copy (l = (newC.l + "d"))
```

---

**Note:** If mutability isn't an issue at all then in the case of data classes a single constructor can be used by making the property var instead of val and in the init block reassigning the property to a mutable instance

---

## Enums

All enums are supported, provided they are annotated with `@CordaSerializable`. Corda supports interoperability of enumerated type versions. This allows such types to be changed over time without breaking backward (or forward) compatibility. The rules and mechanisms for doing this are discussed in [Enum Evolution](#).

## Exceptions

The following rules apply to supported `Throwable` implementations.

1. If you wish for your exception to be serializable and transported type safely it should inherit from either `CordaException` or `CordaRuntimeException`
2. If not, the `Throwable` will deserialize to a `CordaRuntimeException` with the details of the original `Throwable` contained within it, including the class name of the original `Throwable`

## Kotlin Objects

Kotlin's non-anonymous `object`s (i.e. constructs like `object foo : Contract { ... }`) are singletons and treated differently. They are recorded into the stream with no properties, and deserialize back to the singleton instance. Currently, the same is not true of Java singletons, which will deserialize to new instances of the class. This is hard to fix because there's no perfectly standard idiom for Java singletons.

Kotlin's anonymous `object`s (i.e. constructs like `object : Contract { ... }`) are not currently supported and will not serialize correctly. They need to be re-written as an explicit class declaration.

### 13.1.6 Class synthesis

Corda serialization supports dynamically synthesising classes from the supplied schema when deserializing, without the supporting classes being present on the classpath. This can be useful where generic code might expect to be able to use reflection over the deserialized data, for scripting languages that run on the JVM, and also for ensuring classes not on the classpath can be deserialized without loading potentially malicious code.

If the original class implements some interfaces then the carpenter will make sure that all of the interface methods are backed by fields. If that's not the case then an exception will be thrown during deserialization. This check can be turned off with `SerializationContext.withLenientCarpenter`. This can be useful if only the field getters are needed, say in an object viewer.

## Calculated values

In some cases, for example the `exitKeys` field in `FungibleState`, a property in an interface may normally be implemented as a *calculated* value, with a “getter” method for reading it but neither a corresponding constructor parameter nor a “setter” method for writing it. In this case, it will not automatically be included among the properties to be serialized, since the receiving class would ordinarily be able to re-calculate it on demand. However, a synthesized class will not have the method implementation which knows how to calculate the value, and a cast to the interface will fail because the property is not serialized and so the “getter” method present in the interface will not be synthesized.

The solution is to annotate the method with the `SerializableCalculatedProperty` annotation, which will cause the value exposed by the method to be read and transmitted during serialization, but discarded during normal deserialization. The synthesized class will then include a backing field together with a “getter” for the serialized calculated value, and will remain compatible with the interface.

If the annotation is added to the method in the *interface*, then all implementing classes must calculate the value and none may have a corresponding backing field; alternatively, it can be added to the overriding method on each implementing class where the value is calculated and there is no backing field. If the field is a Kotlin `val`, then the annotation should be targeted at its getter method, e.g. `@get:SerializableCalculatedProperty`.

## Future enhancements

Possible future enhancements include:

1. Java singleton support. We will add support for identifying classes which are singletons and identifying the static method responsible for returning the singleton instance
2. Instance internalizing support. We will add support for identifying classes that should be resolved against an instances map to avoid creating many duplicate instances that are equal (similar to `String.intern()`)

### 13.1.7 Type Evolution

Type evolution is the mechanism by which classes can be altered over time yet still remain serializable and deserializable across all versions of the class. This ensures an object serialized with an older idea of what the class “looked like” can be deserialized and a version of the current state of the class instantiated.

More detail can be found in [Default Class Evolution](#).

## 13.2 Pluggable Serializers for CorDapps

### Contents

- *Pluggable Serializers for CorDapps*
  - *Serializer Location*
  - *Writing a Custom Serializer*
  - *Example*
  - *The Proxy Object*
  - *Whitelisting*

To be serializable by Corda Java classes must be compiled with the `-parameters` switch to enable matching of its properties to constructor parameters. This is important because Corda’s internal AMQP serialization scheme will only construct objects using their constructors. However, when recompilation isn’t possible, or classes are built in such a way that they cannot be easily modified for simple serialization, CorDapps can provide custom proxy serializers that Corda can use to move from types it cannot serialize to an interim representation that it can with the transformation to and from this proxy object being handled by the supplied serializer.

### 13.2.1 Serializer Location

Custom serializer classes should follow the rules for including classes found in [Building and installing a CorDapp](#)

### 13.2.2 Writing a Custom Serializer

Serializers must

- Inherit from `net.corda.core.serialization.SerializationCustomSerializer`
- Provide a proxy class to transform the object to and from
- Implement the `toProxy` and `fromProxy` methods
- Be either included into the CorDapp Jar or made known to the running process via the `amqp.custom.serialization.scanSpec` system property. This system property may be necessary to be able to discover custom serializer in the classpath. At a minimum the value of the property should include comma separated set of packages where custom serializers located. Full syntax includes scanning specification as defined by:  
`<http://github.com/lukehutch/fast-classpath-scanner/wiki/2.-Constructor#scan-spec>`

Serializers inheriting from `SerializationCustomSerializer` have to implement two methods and two types.

### 13.2.3 Example

Consider the following class:

```
public final class Example {
    private final Int a
    private final Int b

    // Because this is marked private the serialization framework will not
    // consider it when looking to see which constructor should be used
    // when serializing instances of this class.
    private Example(Int a, Int b) {
        this.a = a;
        this.b = b;
    }

    public static Example of (int[] a) { return Example(a[0], a[1]); }

    public int getA() { return a; }
    public int getB() { return b; }
}
```

Without a custom serializer we cannot serialize this class as there is no public constructor that facilitates the initialisation of all of its properties.

---

**Note:** This is clearly a contrived example, simply making the constructor public would alleviate the issues. However, for the purposes of this example we are assuming that for external reasons this cannot be done.

---

To be serializable by Corda this would require a custom serializer to be written that can transform the unserializable class into a form we can serialize. Continuing the above example, this could be written as follows:

```
/**
 * The class lacks a public constructor that takes parameters it can associate
 * with its properties and is thus not serializable by the CORDA serialization
 * framework.
 */
class Example {
    private int a;
    private int b;

    public int getA() { return a; }
    public int getB() { return b; }

    public Example(List<int> l) {
        this.a = l.get(0);
        this.b = l.get(1);
    }
}

/**
 * This is the class that will Proxy instances of Example within the serializer
 */
public class ExampleProxy {
    /**
     * These properties will be serialized into the byte stream, this is where we
     →choose how to
}
```

(continues on next page)

(continued from previous page)

```

    * represent instances of the object we're proxying. In this example, which is ↵
    ↵somewhat
    * contrived, this choice is obvious. In your own classes / 3rd party libraries, ↵
    ↵however, this
    * may require more thought.
    */
private int proxiedA;
private int proxiedB;

/**
 * The proxy class itself must be serializable by the framework, it must thus ↵
have a constructor that
 * can be mapped to the properties of the class via getter methods.
*/
public int getProxiedA() { return proxiedA; }
public int getProxiedB() { return proxiedB; }

public ExampleProxy(int proxiedA, int proxiedB) {
    this.proxiedA = proxiedA;
    this.proxiedB = proxiedB;
}
}

/**
 * Finally this is the custom serializer that will automatically loaded into the ↵
serialization
 * framework when the CorDapp Jar is scanned at runtime.
*/
public class ExampleSerializer implements SerializationCustomSerializer<Example, ↵
ExampleProxypublic ExampleProxy toProxy(Example obj) {
    return new ExampleProxy(obj.getA(), obj.getB());
}

    /**
     * Conversely, given an instance of the proxy object, revert that back to an ↵
instance of the
     * type being proxied.
     *
     * Essentially convert ExampleProxy -> Example
     */
public Example fromProxy(ExampleProxy proxy) {
    List<int> l = new ArrayList<int>(2);
    l.add(proxy.getProxiedA());
    l.add(proxy.getProxiedB());
    return new Example(l);
}
}
}

```

```

class ExampleSerializer : SerializationCustomSerializer<Example, ExampleSerializer> {
    /**
     * This is the actual proxy class that is used as an intermediate representation
     * of the Example class
     */
    data class Proxy(val a: Int, val b: Int)

    /**
     * This method should be able to take an instance of the type being proxied and
     * transpose it into that form, instantiating an instance of the Proxy object (it
     * is this class instance that will be serialized into the byte stream.
     */
    override fun toProxy(obj: Example) = Proxy(obj.a, obj.b)

    /**
     * This method is used during deserialization. The bytes will have been read
     * from the serialized blob and an instance of the Proxy class returned, we must
     * now be able to transform that back into an instance of our original class.
     *
     * In our example this requires us to evoke the static "of" method on the
     * Example class, transforming the serialized properties of the Proxy instance
     * into a form expected by the construction method of Example.
     */
    override fun fromProxy(proxy: Proxy) : Example {
        val constructorArg = IntArray(2);
        constructorArg[0] = proxy.a
        constructorArg[1] = proxy.b
        return Example.of(constructorArg)
    }
}

```

In the above examples

- ExampleSerializer is the actual serializer that will be loaded by the framework to serialize instances of the Example type.
- ExampleSerializer.Proxy, in the Kotlin example, and ExampleProxy in the Java example, is the intermediate representation used by the framework to represent instances of Example within the wire format.

### 13.2.4 The Proxy Object

The proxy object should be thought of as an intermediate representation that the serialization framework can reason about. One is being written for a class because, for some reason, that class cannot be introspected successfully but that framework. It is therefore important to note that the proxy class must only contain elements that the framework can reason about.

The proxy class itself is distinct from the proxy serializer. The serializer must refer to the unserializable type in the `toProxy` and `fromProxy` methods.

For example, the first thought a developer may have when implementing a proxy class is to simply *wrap* an instance of the object being proxied. This is shown below

```

class ExampleSerializer : SerializationCustomSerializer<Example, ExampleSerializer> {
    /**
     * In this example, we are trying to wrap the Example type to make it serializable
     */

```

(continues on next page)

(continued from previous page)

```
/*
data class Proxy(val e: Example)

override fun toProxy(obj: Example) = Proxy(obj)

override fun fromProxy(proxy: Proxy) : Example {
    return proxy.e
}
```

However, this will not work because what we've created is a recursive loop whereby synthesising a serializer for the `Example` type requires synthesising one for `ExampleSerializer.Proxy`. However, that requires one for `Example` and so on and so forth until we get a `StackOverflowException`.

The solution, as shown initially, is to create the intermediate form (the Proxy object) purely in terms the serialization framework can reason about.

**Important:** When composing a proxy object for a class be aware that everything within that structure will be written into the serialized byte stream.

### 13.2.5 Whitelisting

By writing a custom serializer for a class it has the effect of adding that class to the whitelist, meaning such classes don't need explicitly adding to the CorDapp's whitelist.

### 13.3 Default Class Evolution

Contents

- *Default Class Evolution*
    - *Adding Nullable Properties*
    - *Adding Non Nullable Properties*
      - \* *Constructor Versioning*
    - *Removing Properties*
    - *Reordering Constructor Parameter Order*

Whilst more complex evolutionary modifications to classes require annotating, Corda's serialization framework supports several minor modifications to classes without any external modification save the actual code changes. These are:

1. Adding nullable properties
  2. Adding non nullable properties if, and only if, an annotated constructor is provided
  3. Removing properties
  4. Reordering constructor parameters

### 13.3.1 Adding Nullable Properties

The serialization framework allows nullable properties to be freely added. For example:

```
// Initial instance of the class
data class Example1 (val a: Int, b: String) // (Version A)

// Class post addition of property c
data class Example1 (val a: Int, b: String, c: Int?) // (Version B)
```

A node with version A of class Example1 will be able to deserialize a blob serialized by a node with it at version B as the framework would treat it as a removed property.

A node with the class at version B will be able to deserialize a serialized version A of Example1 without any modification as the property is nullable and will thus provide null to the constructor.

### 13.3.2 Adding Non Nullable Properties

If a non null property is added, unlike nullable properties, some additional code is required for this to work. Consider a similar example to our nullable example above

```
// Initial instance of the class
data class Example2 (val a: Int, b: String) // (Version A)

// Class post addition of property c
data class Example1 (val a: Int, b: String, c: Int) { // (Version B)
    @DeprecatedConstructorForDeserialization(1)
    constructor (a: Int, b: String) : this(a, b, 0) // 0 has been determined as a
    //sensible default
}
```

For this to work we have had to add a new constructor that allows nodes with the class at version B to create an instance from serialised form of that class from an older version, in this case version A as per our example above. A sensible default for the missing value is provided for instantiation of the non null property.

---

**Note:** The `@DeprecatedConstructorForDeserialization` annotation is important, this signifies to the serialization framework that this constructor should be considered for building instances of the object when evolution is required.

Furthermore, the integer parameter passed to the constructor if the annotation indicates a precedence order, see the discussion below.

---

As before, instances of the class at version A will be able to deserialize serialized forms of example B as it will simply treat them as if the property has been removed (as from its perspective, they will have been).

#### Constructor Versioning

If, over time, multiple non nullable properties are added, then a class will potentially have to be able to deserialize a number of different forms of the class. Being able to select the correct constructor is important to ensure the maximum information is extracted.

Consider this example:

```
// The original version of the class
data class Example3 (val a: Int, val b: Int)
```

```
// The first alteration, property c added
data class Example3 (val a: Int, val b: Int, val c: Int)
```

```
// The second alteration, property d added
data class Example3 (val a: Int, val b: Int, val c: Int, val d: Int)
```

```
// The third alteration, and how it currently exists, property e added
data class Example3 (val a: Int, val b: Int, val c: Int, val d: Int, val: Int e) {
    // NOTE: version number purposefully omitted from annotation for demonstration purposes
    @DeprecatedConstructorForDeserialization
    constructor (a: Int, b: Int) : this(a, b, -1, -1, -1) // alt constructor 1
    @DeprecatedConstructorForDeserialization
    constructor (a: Int, b: Int, c: Int) : this(a, b, c, -1, -1) // alt constructor 2
    @DeprecatedConstructorForDeserialization
    constructor (a: Int, b: Int, c: Int, d: Int) : this(a, b, c, d, -1) // alt constructor 3
}
```

In this case, the deserializer has to be able to deserialize instances of class Example3 that were serialized as, for example:

```
Example3 (1, 2)           // example I
Example3 (1, 2, 3)         // example II
Example3 (1, 2, 3, 4)      // example III
Example3 (1, 2, 3, 4, 5)   // example IV
```

Examples I, II, and III would require evolution and thus selection of constructor. Now, with no versioning applied there is ambiguity as to which constructor should be used. For example, example II could use ‘alt constructor 2’ which matches its arguments most tightly or ‘alt constructor 1’ and not instantiate parameter c.

`constructor (a: Int, b: Int, c: Int) : this(a, b, c, -1, -1)`

or

`constructor (a: Int, b: Int) : this(a, b, -1, -1, -1)`

Whilst it may seem trivial which should be picked, it is still ambiguous, thus we use a versioning number in the constructor annotation which gives a strict precedence order to constructor selection. Therefore, the proper form of the example would be:

```
// The third alteration, and how it currently exists, property e added
data class Example3 (val a: Int, val b: Int, val c: Int, val d: Int, val: Int e) {
    @DeprecatedConstructorForDeserialization(1)
    constructor (a: Int, b: Int) : this(a, b, -1, -1, -1) // alt constructor 1
    @DeprecatedConstructorForDeserialization(2)
    constructor (a: Int, b: Int, c: Int) : this(a, b, c, -1, -1) // alt constructor 2
    @DeprecatedConstructorForDeserialization(3)
    constructor (a: Int, b: Int, c: Int, d: Int) : this(a, b, c, d, -1) // alt constructor 3
}
```

Constructors are selected in strict descending order taking the one that enables construction. So, deserializing examples I to IV would give us:

```
Example3 (1, 2, -1, -1, -1) // example I
Example3 (1, 2, 3, -1, -1) // example II
Example3 (1, 2, 3, 4, -1) // example III
Example3 (1, 2, 3, 4, 5) // example IV
```

### 13.3.3 Removing Properties

Property removal is effectively a mirror of adding properties (both nullable and non nullable) given that this functionality is required to facilitate the addition of properties. When this state is detected by the serialization framework, properties that don't have matching parameters in the main constructor are simply omitted from object construction.

```
// Initial instance of the class
data class Example4 (val a: Int?, val b: String?, val c: Int?) // (Version A)

// Class post removal of property 'a'
data class Example4 (val b: String?, c: Int?) // (Version B)
```

In practice, what this means is removing nullable properties is possible. However, removing non nullable properties isn't because a node receiving a message containing a serialized form of an object with fewer properties than it requires for construction has no capacity to guess at what values should or could be used as sensible defaults. When those properties are nullable it simply sets them to null.

### 13.3.4 Reordering Constructor Parameter Order

Properties (in Kotlin this corresponds to constructor parameters) may be reordered freely. The evolution serializer will create a mapping between how a class was serialized and its current constructor parameter order. This is important to our AMQP framework as it constructs objects using their primary (or annotated) constructor. The ordering of whose parameters will have determined the way an object's properties were serialised into the byte stream.

For an illustrative example consider a simple class:

```
data class Example5 (val a: Int, val b: String)

val e = Example5(999, "hello")
```

When we serialize `e` its properties will be encoded in order of its primary constructors parameters, so:

`999,hello`

Were those parameters to be reordered post serialisation then deserializing, without evolution, would fail with a basic type error as we'd attempt to create the new value of `Example5` with the values provided in the wrong order:

```
// changed post serialisation
data class Example5 (val b: String, val a: Int)
```

```
| 999 | hello | <--- Extract properties to pass to constructor from byte stream
|     |
|     +-----+
+-----+     |
|           |
deserializedValue = Example5(999, "hello") <--- Resulting attempt at construction
```

(continues on next page)

(continued from previous page)

```

|           |
|           |
|           |
|           |
|           |   <--- Will clearly fail as 999 is not a
|           |   string and hello is not an integer
|           |
data class Example5 (val b: String, val a: Int)

```

## 13.4 Enum Evolution

### Contents

- *Enum Evolution*
  - *The Purpose of Annotating Changes*
  - *Evolution Transmission*
  - *Evolution Precedence*
  - *Renaming Constants*
    - \* *Rules*
  - *Adding Constants*
    - \* *Rules*
  - *Combining Evolutions*
  - *Unsupported Evolutions*

In the continued development of a CorDapp an enumerated type that was fit for purpose at one time may require changing. Normally, this would be problematic as anything serialised (and kept in a vault) would run the risk of being unable to be deserialized in the future or older versions of the app still alive within a compatibility zone may fail to deserialize a message.

To facilitate backward and forward support for alterations to enumerated types Corda's serialization framework supports the evolution of such types through a well defined framework that allows different versions to interoperate with serialised versions of an enumeration of differing versions.

This is achieved through the use of certain annotations. Whenever a change is made, an annotation capturing the change must be added (whilst it can be omitted any interoperability will be lost). Corda supports two modifications to enumerated types, adding new constants, and renaming existing constants

**Warning:** Once added evolution annotations MUST NEVER be removed from a class, doing so will break both forward and backward compatibility for this version of the class and any version moving forward

### 13.4.1 The Purpose of Annotating Changes

The biggest hurdle to allowing enum constants to be changed is that there will exist instances of those classes, either serialized in a vault or on nodes with the old, unmodified, version of the class that we must be able to interoperate with. Thus if a received data structure references an enum assigned a constant value that doesn't exist on the running JVM, a solution is needed.

For this, we use the annotations to allow developers to express their backward compatible intentions.

In the case of renaming constants this is somewhat obvious, the deserializing node will simply treat any constants it doesn't understand as their "old" values, i.e. those values that it currently knows about.

In the case of adding new constants the developer must chose which constant (that existed *before* adding the new one) a deserializing system should treat any instances of the new one as.

---

**Note:** Ultimately, this may mean some design compromises are required. If an enumeration is planned as being often extended and no sensible defaults will exist then including a constant in the original version of the class that all new additions can default to may make sense

---

### 13.4.2 Evolution Transmission

An object serializer, on creation, will inspect the class it represents for any evolution annotations. If a class is thus decorated those rules will be encoded as part of any serialized representation of a data structure containing that class. This ensures that on deserialization the deserializing object will have access to any transformative rules it needs to build a local instance of the serialized object.

### 13.4.3 Evolution Precedence

On deserialization (technically on construction of a serialization object that facilitates serialization and deserialization) a class's fingerprint is compared to the fingerprint received as part of the AMQP header of the corresponding class. If they match then we are sure that the two class versions are functionally the same and no further steps are required save the deserialization of the serialized information into an instance of the class.

If, however, the fingerprints differ then we know that the class we are attempting to deserialize is different than the version we will be deserializing it into. What we cannot know is which version is newer, at least not by examining the fingerprint

---

**Note:** Corda's AMQP fingerprinting for enumerated types include the type name and the enum constants

---

Newer vs older is important as the deserializer needs to use the more recent set of transforms to ensure it can transform the serialised object into the form as it exists in the deserializer. Newness is determined simply by length of the list of all transforms. This is sufficient as transform annotations should only ever be added

**Warning:** technically there is nothing to prevent annotations being removed in newer versions. However, this will break backward compatibility and should thus be avoided unless a rigorous upgrade procedure is in place to cope with all deployed instances of the class and all serialised versions existing within vaults.

Thus, on deserialization, there will be two options to chose from in terms of transformation rules

1. Determined from the local class and the annotations applied to it (the local copy)
2. Parsed from the AMQP header (the remote copy)

Which set is used will simply be the largest.

### 13.4.4 Renaming Constants

Renamed constants are marked as such with the `@CordaSerializationTransformRenames` meta annotation that wraps a list of `@CordaSerializationTransformRename` annotations. Each rename requiring an instance in the list.

Each instance must provide the new name of the constant as well as the old. For example, consider the following enumeration:

```
enum class Example {
    A, B, C
}
```

If we were to rename constant C to D this would be done as follows:

```
@CordaSerializationTransformRenames (
    CordaSerializationTransformRename("D", "C")
)
enum class Example {
    A, B, D
}
```

---

**Note:** The parameters to the `CordaSerializationTransformRename` annotation are defined as ‘to’ and ‘from’, so in the above example it can be read as constant D (given that is how the class now exists) was renamed from C

---

In the case where a single rename has been applied the meta annotation may be omitted. Thus, the following is functionally identical to the above:

```
@CordaSerializationTransformRename("D", "C")
enum class Example {
    A, B, D
}
```

However, as soon as a second rename is made the meta annotation must be used. For example, if at some time later B is renamed to E:

```
@CordaSerializationTransformRenames (
    CordaSerializationTransformRename(from = "B", to = "E"),
    CordaSerializationTransformRename(from = "C", to = "D")
)
enum class Example {
    A, E, D
}
```

### Rules

1. A constant cannot be renamed to match an existing constant, this is enforced through language constraints
2. A constant cannot be renamed to a value that matches any previous name of any other constant

If either of these covenants are inadvertently broken, a `NotSerializableException` will be thrown on detection by the serialization engine as soon as they are detected. Normally this will be the first time an object doing so is serialized. However, in some circumstances, it could be at the point of deserialization.

### 13.4.5 Adding Constants

Enumeration constants can be added with the `@CordaSerializationTransformEnumDefaults` meta annotation that wraps a list of `CordaSerializationTransformEnumDefault` annotations. For each constant added an annotation must be included that signifies, on deserialization, which constant value should be used in place of the serialised property if that value doesn't exist on the version of the class as it exists on the deserializing node.

```
enum class Example {
    A, B, C
}
```

If we were to add the constant D

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("D", "C")
)
enum class Example {
    A, B, C, D
}
```

---

**Note:** The parameters to the `CordaSerializationTransformEnumDefault` annotation are defined as 'new' and 'old', so in the above example it can be read as constant D should be treated as constant C if you, the deserializing node, don't know anything about constant D

---

**Note:** Just as with the `CordaSerializationTransformRename` transformation if a single transform is being applied then the meta transform may be omitted.

```
@CordaSerializationTransformEnumDefault("D", "C")
enum class Example {
    A, B, C, D
}
```

---

New constants may default to any other constant older than them, including constants that have also been added since inception. In this example, having added D (above) we add the constant E and chose to default it to D

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("E", "D"),
    CordaSerializationTransformEnumDefault("D", "C")
)
enum class Example {
    A, B, C, D, E
}
```

---

**Note:** Alternatively, we could have decided both new constants should have been defaulted to the first element

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("E", "A"),
    CordaSerializationTransformEnumDefault("D", "A")
)
enum class Example {
    A, B, C, D, E
}
```

When deserializing the most applicable transform will be applied. Continuing the above example, deserializing nodes could have three distinct views on what the enum Example looks like (annotations omitted for brevity)

```
// The original version of the class. Will deserialize: -
//   A -> A
//   B -> B
//   C -> C
//   D -> C
//   E -> C
enum class Example {
    A, B, C
}
```

```
// The class as it existed after the first addition. Will deserialize:
//   A -> A
//   B -> B
//   C -> C
//   D -> D
//   E -> D
enum class Example {
    A, B, C, D
}
```

```
// The current state of the class. All values will deserialize as themselves
enum class Example {
    A, B, C, D, E
}
```

Thus, when deserializing a value that has been encoded as E could be set to one of three constants (E, D, and C) depending on how the deserializing node understands the class.

## Rules

1. New constants must be added to the end of the existing list of constants
2. Defaults can only be set to “older” constants, i.e. those to the left of the new constant in the list
3. Constants must never be removed once added
4. New constants can be renamed at a later date using the appropriate annotation
5. When renamed, if a defaulting annotation refers to the old name, it should be left as is

### 13.4.6 Combining Evolutions

Renaming constants and adding constants can be combined over time as a class changes freely. Added constants can in turn be renamed and everything will continue to be deserializeable. For example, consider the following enum:

```
enum class OngoingExample { A, B, C }
```

For the first evolution, two constants are added, D and E, both of which are set to default to C when not present

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("E", "C"),
    CordaSerializationTransformEnumDefault("D", "C")
)
enum class OngoingExample { A, B, C, D, E }
```

Then lets assume constant C is renamed to CAT

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("E", "C"),
    CordaSerializationTransformEnumDefault("D", "C")
)
@CordaSerializationTransformRename("C", "CAT")
enum class OngoingExample { A, B, CAT, D, E }
```

Note how the first set of modifications still reference C, not CAT. This is as it should be and will continue to work as expected.

Subsequently is is fine to add an additional new constant that references the renamed value.

```
@CordaSerializationTransformEnumDefaults (
    CordaSerializationTransformEnumDefault("F", "CAT"),
    CordaSerializationTransformEnumDefault("E", "C"),
    CordaSerializationTransformEnumDefault("D", "C")
)
@CordaSerializationTransformRename("C", "CAT")
enum class OngoingExample { A, B, CAT, D, E, F }
```

### 13.4.7 Unsupported Evolutions

The following evolutions are not currently supports

1. Removing constants
2. Reordering constants

## 13.5 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 Corda 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 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 synthesize any classes found in the blob that aren't on the classpath.

### 13.5.1 Supported formats

The inspector can read **input data** in three formats: raw binary, hex encoded text and base64 encoded text. For instance if you have retrieved your binary data and it looks like this:

```
636f7264610100000080c562000000000001d000003072000000300a3226e65742e636f7264613a38674f537471464b414a  
↔...
```

then you have hex encoded data. If it looks like this it's base64 encoded:

```
Y29yZGEAAAAgMViAAAAAAAB0AAAMHIAAAADAKMibmV0LmNvcnRhOjhNT1N0cUZLQUpQVWVvY2Z2M1N1U1E9PdAAACc1AAAAAgCj...  
↔...
```

And if it looks like something vomited over your screen it's raw binary. You don't normally need to care about these differences because the tool will try every format until it works.

Something that's useful to know about Corda's format is that it always starts with the word "corda" in binary. Try hex decoding 636f726461 using the [online hex decoder tool here](#) to see for yourself.

**Output data** can be in either a slightly extended form of YAML or JSON. YAML (Yet another markup language) is a bit easier to read for humans and is the default. JSON can of course be parsed by any JSON library in any language.

---

**Note:** 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.

---

### 13.5.2 Example

Here's what a node-info file from the node's data directory may look like:

- 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 |-
      VFRy4frbgRDbCpK1Vo88PyUojo1vbRnMR3ROR2abTFk7yJ14901aeScX/
      ↔CiEP+CDGiMRsdw01cXt\nhKSobAY7Dw==
```

- JSON:

```
net.corda.nodeapi.internal.SignedNodeInfo
{
  "raw" : {
    "class" : "net.corda.core.node.NodeInfo",
```

(continues on next page)

(continued from previous page)

```

"deserialized" : {
    "addresses" : [ "localhost:10005" ],
    "legalIdentitiesAndCerts" : [ "O=BankOfCorda, L=London, C=GB" ],
    "platformVersion" : 4,
    "serial" : 1527851068715
},
"signatures" : [ "VFRy4frbgRDbCpK1Vo88PyUoj01vbRnMR3ROR2abTFk7yJ14901aeScX/
←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.

### 13.5.3 Command-line options

The blob inspector can be started with the following command-line options:

```
blob-inspector [-hv] [--full-parties] [--schema] [--format=type]
                [--input-format=type] [--logging-level=<loggingLevel>] SOURCE
                [COMMAND]
```

- `--format=type`: Output format. Possible values: [YAML, JSON]. Default: YAML.
- `--input-format=type`: Input format. If the file can't be decoded with the given value it's auto-detected, so you should never normally need to specify this. Possible values [BINARY, HEX, BASE64]. Default: BINARY.
- `--full-parties`: Display the `owningKey` and `certPath` properties of `Party` and `PartyAndReference` objects respectively.
- `--schema`: Print the blob's schema first.
- `--verbose`, `--log-to-console`, `-v`: If set, prints logging to the console as well as to a file.
- `--logging-level=<loggingLevel>`: Enable logging at this level and higher. Possible values: ERROR, WARN, INFO, DEBUG, TRACE. Default: INFO.
- `--help`, `-h`: Show this help message and exit.
- `--version`, `-V`: Print version information and exit.

#### Sub-commands

`install-shell-extensions`: Install `blob-inspector` alias and auto completion for bash and zsh. See [Shell extensions for CLI Applications](#) for more info.

---

CHAPTER  
FOURTEEN

---

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 “[Interacting with a node](#)”) then your JSON mapper can resolve identities found in objects.

The API is described in detail here:

- [Kotlin API docs](#)
- [JavaDoc](#)

```
import net.corda.jackson.JacksonSupport

val mapper = JacksonSupport.createNonRpcMapper()
val json = mapper.writeValueAsString(myCordaState) // myCordaState can be any object.
```

```
import net.corda.jackson.JacksonSupport

ObjectMapper mapper = JacksonSupport.createNonRpcMapper()
String json = mapper.writeValueAsString(myCordaState) // myCordaState can be any
↪object.
```

---

**Note:** The way mappers interact with identity and RPC is likely to change in a future release.

---

---

**CHAPTER  
FIFTEEN**

---

## **TROUBLESHOOTING**

Please report any issues on our StackOverflow page: <https://stackoverflow.com/questions/tagged/corda>.

## NODES

### 16.1 Node folder structure

A folder containing a Corda node files has the following structure:

```
.  
├── additional-node-infos    // Additional node infos to load into the network map  
└── cache, beyond what the network map server provides  
├── artemis                  // Stores buffered P2P messages  
├── brokers                  // Stores buffered RPC messages  
├── certificates             // The node's certificates  
├── corda-webserver.jar      // The built-in node webserver (DEPRECATED)  
├── corda.jar                // The core Corda libraries (This is the actual Corda  
└── node implementation)  
├── cordapps                 // The CorDapp JARs installed on the node  
├── drivers                  // Contains a Jolokia driver used to export JMX metrics,  
└── the node loads any additional JAR files from this directory at startup.  
├── logs                     // The node's logs  
├── network-parameters        // The network parameters automatically downloaded from  
└── the network map server  
├── node.conf                // The node's configuration files  
├── persistence.mv.db         // The node's database  
└── shell-commands            // Custom shell commands defined by the node owner
```

You install CorDapps on the node by placing CorDapp JARs in the `cordapps` folder.

In development mode (i.e. when `devMode = true`), the `certificates` directory is filled with pre-configured keystores if they do not already exist to ensure that developers can get the nodes working as quickly as possible.

**Warning:** These pre-configured keystores are not secure and must not be used in a production environments.

The keystores store the key pairs and certificates under the following aliases:

- `nodekeystore.jks` uses the aliases `cordaclientca` and `identity-private-key`
- `sslkeystore.jks` uses the alias `cordaclienttls`

All the keystores use the password provided in the node's configuration file using the `keyStorePassword` attribute. If no password is configured, it defaults to `cordacadevpass`.

To learn more, see [Network certificates](#).

## 16.2 Node identity

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)

Note that the serial number is intentionally excluded from Corda certificates in order to minimise scope for uncertainty in the distinguished name format. The distinguished name qualifier has been removed due to technical issues; consideration was given to “Corda” as qualifier, however the qualifier needs to reflect the Corda compatibility zone, not the technology involved. There may be many Corda namespaces, but only one R3 namespace on Corda. The ordering of attributes is important.

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*<[https://en.wikipedia.org/wiki/ISO\\_3166-1\\_alpha-2](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2)> two letter code in upper-case
- **The organisation field of the name obeys the following constraints:**
  - Has at least two letters
  - 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
  - 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.

---

**Note:** The network operator of a Corda Network may put additional constraints on node naming in place.

---

### 16.2.1 External identifiers

Mappings to external identifiers such as Companies House nos., 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.

## 16.3 Node configuration

### Contents

- *Node configuration*
  - *Configuration file location*
  - *Configuration file format*
  - *Overriding values from node.conf*
  - *Configuration file fields*
  - *Reference.conf*
  - *Configuration examples*
    - \* *Node configuration hosting the IRSDemo services*
    - \* *Simple notary configuration file*
    - \* *Node configuration with diffrent URL for NetworkMap and Doorman*

### 16.3.1 Configuration file location

When starting a node, the `corda.jar` file defaults to reading the node's configuration from a `node.conf` file in the directory from which the command to launch Corda is executed. There are two command-line options to override this behaviour:

- The `--config-file` command line option allows you to specify a configuration file with a different name, or in a different file location. Paths are relative to the current working directory
- The `--base-directory` command line option allows you to specify the node's workspace location. A `node.conf` configuration file is then expected in the root of this workspace.

If you specify both command line arguments at the same time, the node will fail to start.

### 16.3.2 Configuration file format

The Corda configuration file uses the HOCON format which is a superset of JSON. Please visit <https://github.com/typesafehub/config/blob/master/HOCON.md> for further details.

Please do NOT use double quotes ("") in configuration keys.

Node setup will log Config files should not contain " in property names. Please fix: [key] as an error when it finds double quotes around keys. This prevents configuration errors when mixing keys containing . wrapped with double quotes and without them e.g.: The property "dataSourceProperties.dataSourceClassName" = "val" in *Reference.conf* would be not overwritten by the property dataSourceProperties.dataSourceClassName = "val2" in *node.conf*.

By default the node will fail to start in presence of unknown property keys. To alter this behaviour, the on-unknown-config-keys command-line argument can be set to IGNORE (default is FAIL).

### 16.3.3 Overriding values from node.conf

**Environment variables** For example: \${NODE\_TRUST\_STORE\_PASSWORD} would be replaced by the contents of environment variable NODE\_TRUST\_STORE\_PASSWORD (see: [Logging](#) section).

**JVM options** JVM options or environmental variables prefixed with corda. can override node.conf fields.

Provided system properties can also set values for absent fields in node.conf. This is an example of adding/overriding the keyStore password :

```
java -Dcorda.rpcSettings.ssl.keyStorePassword=mypassword -jar node.jar
```

### 16.3.4 Configuration file fields

---

**Note:** The available configuration fields are listed below in alphabetic order.

---

**additionalP2PAddresses** An array of additional host:port values, which will be included in the advertised NodeInfo in the network map in addition to the *p2pAddress*. Nodes can use this configuration option to advertise HA endpoints and aliases to external parties.

*Default:* empty list

**attachmentContentCacheSizeMegaBytes** Optionally specify how much memory should be used to cache attachment contents in memory.

*Default:* 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.

*Default:* 1024

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

**Important: old configuration value, please use networkServices**

*Default:* not defined

**cordappSignerKeyFingerprintBlacklist** List of the public keys fingerprints (SHA-256 of public key hash) not allowed as Cordapp JARs signers. The node will not load Cordapps signed by those keys. The option takes effect only in production mode and defaults to Corda development keys ([ "56CA54E803CB87C8472EBD3FBC6A2F1876E814CEEBF74860BD46997F40729367", "83088052AF16700457AE2C978A7D8AC38DD6A7C713539D00B897CD03A5E5D31D" ]), in development mode any key is allowed to sign Cordpapp JARs.

*Default:* not defined

**crlCheckSoftFail** This is a boolean flag that when enabled (i.e. `true` value is set) causes certificate revocation list (CRL) checking to use soft fail mode. Soft fail mode allows the revocation check to succeed if the revocation status cannot be determined because of a network error. If this parameter is set to `false` rigorous CRL checking takes place. This involves each certificate in the certificate path being checked for a CRL distribution point extension, and that this extension points to a URL serving a valid CRL. This means that if any CRL URL in the certificate path is inaccessible, the connection with the other party will fail and be marked as bad. Additionally, if any certificate in the hierarchy, including the self-generated node SSL certificate, is missing a valid CRL URL, then the certificate path will be marked as invalid.

*Default:* true

**custom** Set custom command line attributes (e.g. Java system properties) on the node process via the capsule launcher

**jvmArgs:** A list of JVM arguments to apply to the node process. This removes any defaults specified from `corda.jar`, but can be overridden from the command line. See [Setting JVM arguments](#) for examples and details on the precedence of the different approaches to settings arguments.

*Default:* not defined

**database** Database configuration

**transactionIsolationLevel:** Transaction isolation level as defined by the `TRANSACTION_` constants in `java.sql.Connection`, but without the `TRANSACTION_` prefix.

*Default:* REPEATABLE\_READ

**exportHibernateJMXStatistics:** Whether to export Hibernate JMX statistics.

**Caution: enabling this option causes expensive run-time overhead**

*Default:* false

**initialiseSchema** Boolean which indicates whether to update the database schema at startup (or create the schema when node starts for the first time). If set to `false` on startup, the node will validate if it's running against a compatible database schema.

*Default:* true

**initialiseAppSchema** The property allows to override `database.initialiseSchema` for the Hibernate DDL generation for CorDapp schemas. `UPDATE` performs an update of CorDapp schemas, while `VALID` only verifies their integrity and `NONE` performs no check. When `initialiseSchema` is set to `false`, then `initialiseAppSchema` may be set as `VALID` or `NONE` only.

*Default:* CorDapp schema creation is controlled with `initialiseSchema`.

**dataSourceProperties** This section is used to configure the JDBC connection and database driver used for the node's persistence. Node database contains example configurations for other database providers. To add additional data source properties (for a specific JDBC driver) use the `dataSource.` prefix with the property name (e.g. `dataSource.customProperty = value`).

**dataSourceClassName** JDBC Data Source class name.

**dataSource.url** JDBC database URL.

**dataSource.user** Database user.

**dataSource.password** Database password.

*Default:*

```
dataSourceClassName = org.h2.jdbcx.JdbcDataSource
dataSource.url = "jdbc:h2:file:${baseDirectory}/persistence;DB_CLOSE_ON_
˓→EXIT=FALSE;WRITE_DELAY=0;LOCK_TIMEOUT=10000"
dataSource.user = sa
dataSource.password = ""
```

**detectPublicIp** This flag toggles the auto IP detection behaviour. If enabled, 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 true to enable.

*Default:* false

**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 the existing schema, failing on node start if the schema is either not present or not compatible. If no value is specified in the node configuration file, the node will attempt to detect if it's running on a developer machine and set devMode=true in that case. This value can be overridden from the command line using the --dev-mode option.

*Default:* Corda will try to establish based on OS environment

**devModeOptions** Allows modification of certain devMode features

**Important: This is an unsupported configuration.**

**allowCompatibilityZone** Allows a node configured to operate in development mode to connect to a compatibility zone.

*Default:* not defined

**emailAddress** The email address responsible for node administration, used by the Compatibility Zone administrator.

*Default:* company@example.com

**extraNetworkMapKeys** An optional list of private network map UUIDs. Your node will fetch the public network and private network maps based on these keys. Private network UUID should be provided by network operator and lets you see nodes not visible on public network.

**Important: This is a temporary feature for onboarding network participants that limits their visibility for privacy reasons.**

*Default:* not defined

**flowMonitorPeriodMillis** Duration of the period suspended flows waiting for IO are logged.

*Default:* 60 seconds

**flowMonitorSuspensionLoggingThresholdMillis** Threshold duration suspended flows waiting for IO need to exceed before they are logged.

*Default:* 60 seconds

**flowTimeout** When a flow implementing the `TimedFlow` interface and setting the `isTimeoutEnabled` flag does not complete within a defined elapsed time, it is restarted from the initial checkpoint. Currently only used for notarisation requests with clustered notaries: if a notary cluster member dies while processing a notarisation request, the client flow eventually times out and gets restarted. On restart the request is resent to a different notary cluster member in a round-robin fashion. Note that the flow will keep retrying forever.

**timeout** The initial flow timeout period.

*Default:* 30 seconds

**maxRestartCount** The number of retries the back-off time keeps growing for. For subsequent retries, the timeout value will remain constant.

*Default:* 6

**backoffBase** The base of the exponential backoff,  $t_{\text{wait}} = \text{timeout} * \text{backoffBase}^{\{retryCount\}}$

*Default:* 1.8

**h2Port (deprecated)** Defines port for h2 DB.

**Important: Deprecated please use h2Setting instead**

**h2Settings** Sets the H2 JDBC server host and port. See [Database access when running H2](#). For non-localhost address the database password needs to be set in `dataSourceProperties`.

*Default:* NULL

**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 = [ ${baseDirectory}/libs ]`. (Note that you have to use the `baseDirectory` substitution value when pointing to a relative path).

*Default:* not defined

**jmxMonitoringHttpPort** If set, will enable JMX metrics reporting via the Jolokia HTTP/JSON agent on the corresponding port. Default Jolokia access url is <http://127.0.0.1:port/jolokia/>

*Default:* not defined

**jmxReporterType** Provides an option for registering an alternative JMX reporter. Available options are `JOLOKIA` and `NEW_RELIC`.

The Jolokia configuration is provided by default. The New Relic configuration leverages the [Dropwizard NewRelicReporter](#) solution. See [Introduction to New Relic for Java](#) for details on how to get started and how to install the New Relic Java agent.

*Default:* `JOLOKIA`

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

**Important: 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.**

*Default:* `cordacadevpass`

**lazyBridgeStart** Internal option.

**Important: Please do not change.**

*Default:* true

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

*Default:* not defined

**messagingServerExternal** If `messagingServerAddress` is specified the default assumption is that the artemis broker is running externally. Setting this to `false` overrides this behaviour and runs the artemis internally to the node, but bound to the address specified in `messagingServerAddress`. This allows the address and

port advertised in `p2pAddress` to differ from the local binding, especially if there is external remapping by firewalls, load balancers , or routing rules. Note that `detectPublicIp` should be set to `false` to ensure that no translation of the `p2pAddress` occurs before it is sent to the network map.

*Default:* not defined

**myLegalName** The legal identity of the node. This acts as a human-readable alias to the node's public key and can be used with the network map to look up the node's info. This is the name that is used in the node's certificates (either when requesting them from the doorman, or when auto-generating them in dev mode). At runtime, Corda checks whether this name matches the name in the node's certificates. For more details please read [Node identity](#) chapter.

*Default:* not defined

**notary** 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.

*Default:* false

**serviceLegalName** If the node is part of a distributed cluster, specify the legal name of the cluster. At runtime, Corda checks whether this name matches the name of the certificate of the notary cluster.

*Default:* not defined

**raft** (*Experimental*) If part of a distributed Raft cluster, specify this configuration object with the following settings:

**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.

*Default:* not defined

**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.

*Default:* not defined

**bftSMaRt** (*Experimental*) If part of a distributed BFT-SMaRt cluster, specify this configuration object with the following settings:

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

*Default:* not defined

**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.

*Default:* not defined

**networkParameterAcceptanceSettings** Optional settings for managing the network parameter auto-acceptance behaviour. If not provided then the defined defaults below are used.

**autoAcceptEnabled** This flag toggles auto accepting of network parameter changes. If a network operator issues a network parameter change which modifies only auto-acceptable options and this behaviour is enabled then the changes will be accepted without any manual intervention from the node operator. See [The network map](#) for more information on the update process and current auto-acceptable parameters. Set to `false` to disable.

*Default:* true

**excludedAutoAcceptableParameters** List of auto-acceptable parameter names to explicitly exclude from auto-accepting. Allows a node operator to control the behaviour at a more granular level.

*Default:* empty list

**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.

**Important: Only one of “compatibilityZoneURL“ or “networkServices“ should be used.**

**doormanURL** Root address of the network registration service.

*Default:* not defined

**networkMapURL** Root address of the network map service.

*Default:* not defined

**pnm** Optional UUID of the private network operating within the compatibility zone this node should be joining.

*Default:* not defined

**p2pAddress** The host and port on which the node is available for protocol operations over ArtemisMQ.

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.

*Default:* not defined

**rpcAddress (deprecated)** 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.

**Important: Deprecated. Use rpcSettings instead.**

*Default:* not defined

**rpcSettings** Options for the RPC server exposed by the Node.

**Important: The RPC SSL certificate is used by RPC clients to authenticate the connection. The Node operator must provide RPC clients with a truststore containing the certificate they can trust. We advise Node operators to not use the P2P keystore for RPC. The node can be run with the “generate-rpc-ssl-settings” command, which generates a secure keystore and truststore that can be used to secure the RPC connection. You can use this if you have no special requirements.**

**address** host and port for the RPC server binding.

*Default:* not defined

**adminAddress** host and port for the RPC admin binding (this is the endpoint that the node process will connect to).

*Default:* not defined

**standAloneBroker** boolean, indicates whether the node will connect to a standalone broker for RPC.

*Default:* false

**useSsl** boolean, indicates whether or not the node should require clients to use SSL for RPC connections.

*Default:* false

**ssl** (mandatory if `useSsl=true`) SSL settings for the RPC server.

**keyStorePath** Absolute path to the key store containing the RPC SSL certificate.

*Default:* not defined

**keyStorePassword** Password for the key store.

*Default:* not defined

**rpcUsers** A list of users who are authorised to access the RPC system. Each user in the list is a configuration object with the following fields:

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

*Default:* not defined

**password** The password

*Default:* not defined

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

*Default:* not defined

**security** Contains various nested fields controlling user authentication/authorization, in particular for RPC accesses. See [Interacting with a node](#) for details.

**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 e.g. `sshd { port = 2222 }`.

*Default:* not defined

**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 }`

*Default:* not defined

**transactionCacheSizeMegaBytes** Optionally specify how much memory should be used for caching of ledger transactions in memory.

*Default:* 8 MB plus 5% of all heap memory above 300MB.

**tlsCertCrlDistPoint** CRL distribution point (i.e. URL) for the TLS certificate. Default value is NULL, which indicates no CRL availability for the TLS certificate.

**Important:** This needs to be set if `crlCheckSoftFail` is false (i.e. strict CRL checking is on).

*Default:* NULL

**tlsCertCrlIssuer** CRL issuer (given in the X500 name format) for the TLS certificate. Default value is NULL, which indicates that the issuer of the TLS certificate is also the issuer of the CRL.

**Important:** If this parameter is set then ‘`tlsCertCrlDistPoint`’ needs to be set as well.

*Default:* NULL

**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.

*Default:* trustpass

**useTestClock** Internal option.

**Important: Please do not change.**

*Default:* false

**verifierType** Internal option.

**Important: Please do not change.**

*Default:* InMemory

### 16.3.5 Reference.conf

A set of default configuration options are loaded from the built-in resource file /node/src/main/resources/reference.conf. This file can be found in the :node gradle module of the Corda repository. Any options you do not specify in your own node.conf file will use these defaults.

Here are the contents of reference.conf file:

```
additionalP2PAddresses = []
crlCheckSoftFail = true
database = {
    transactionIsolationLevel = "REPEATABLE_READ"
    exportHibernateJMXStatistics = "false"
}
dataSourceProperties = {
    dataSourceClassName = org.h2.jdbcx.JdbcDataSource
    dataSource.url = "jdbc:h2:file:${baseDirectory}/persistence;DB_CLOSE_ON_
˓→EXIT=FALSE;WRITE_DELAY=0;LOCK_TIMEOUT=10000"
    dataSource.user = sa
    dataSource.password = ""
}
emailAddress = "admin@company.com"
flowTimeout {
    timeout = 30 seconds
    maxRestartCount = 6
    backoffBase = 1.8
}
jmxReporterType = JOLOKIA
keyStorePassword = "cordacadevpass"
lazyBridgeStart = true
rpcSettings = {
    useSsl = false
    standAloneBroker = false
}
trustStorePassword = "trustpass"
useTestClock = false
verifierType = InMemory
```

### 16.3.6 Configuration examples

#### Node configuration hosting the IRSDemo services

General node configuration file for hosting the IRSDemo services

```

myLegalName = "O=Bank A, L=London, C=GB"
keyStorePassword = "cordacadevpass"
trustStorePassword = "trustpass"
crlCheckSoftFail = true
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"
}
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"
}
notary {
    validating = false
}
devMode = false
networkServices {
    doormanURL = "https://cz.example.com"
    networkMapURL = "https://cz.example.com"
}

```

### Node configuration with different URL for NetworkMap and Doorman

Configuring a node where the Corda Compatibility Zone's registration and Network Map services exist on different URLs

```

myLegalName = "O=Bank A, L=London, C=GB"
keyStorePassword = "cordacadevpass"
trustStorePassword = "trustpass"
crlCheckSoftFail = true
dataSourceProperties {

```

(continues on next page)

(continued from previous page)

```

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"
}
rpcUsers = [
    { username=user1, password=letmein, permissions=[ StartFlow.net.corda.protocols.
    ↵CashProtocol ] }
]
devMode = false
networkServices {
    doormanURL = "https://registration.example.com"
    networkMapURL = "https://cz.example.com"
}

```

## 16.4 Node command-line options

The node can optionally be started with the following command-line options:

- `--base-directory, -b`: The node working directory where all the files are kept (default: `.`).
- `--config-file, -f`: The path to the config file. Defaults to `node.conf`.
- `--dev-mode, -d`: Runs the node in development mode. Unsafe in production. Defaults to true on MacOS and desktop versions of Windows. False otherwise.
- `--no-local-shell, -n`: Do not start the embedded shell locally.
- `--on-unknown-config-keys <[FAIL, INFO]>`: How to behave on unknown node configuration. Defaults to FAIL.
- `--sshd`: Enables SSH server for node administration.
- `--sshd-port`: Sets the port for the SSH server. If not supplied and SSH server is enabled, the port defaults to 2222.
- `--verbose, --log-to-console, -v`: If set, prints logging to the console as well as to a file.
- `--logging-level=<loggingLevel>`: Enable logging at this level and higher. Possible values: ERROR, WARN, INFO, DEBUG, TRACE. Default: INFO.
- `--help, -h`: Show this help message and exit.
- `--version, -V`: Print version information and exit.

### 16.4.1 Sub-commands

`clear-network-cache`: Clears local copy of network map, on node startup it will be restored from server or file system.

initial-registration: Starts initial node registration with the compatibility zone to obtain a certificate from the Doorman.

Parameters:

- `--network-root-truststore`, `-t required`: Network root trust store obtained from network operator.
- `--network-root-truststore-password`, `-p`: Network root trust store password obtained from network operator.

generate-node-info: Performs the node start-up tasks necessary to generate the nodeInfo file, saves it to disk, then exits.

generate-rpc-ssl-settings: Generates the SSL keystore and truststore for a secure RPC connection.

install-shell-extensions: Install corda alias and auto completion for bash and zsh. See [Shell extensions for CLI Applications](#) for more info.

validate-configuration: Validates the actual configuration without starting the node.

## 16.4.2 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.

# 16.5 Node administration

## 16.5.1 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. Hibernate (the JPA provider used by Corda) specific log messages of level `WARN` and above will be logged to the diagnostic log file, which is stored in the same location as other log files (`logs` subdirectory by default). This is because Hibernate may log messages at `WARN` and `ERROR` that are handled internally by Corda and do not need operator attention. If they do, they will be logged by Corda itself in the main node log file.

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 `Log4j2` configuration file assigned to the `log4j.configurationFile` system property.

The node is using `log4j2` asynchronous logging by default (configured via `log4j2` properties file in its resources) to ensure that log message flushing is not slowing down the actual processing. If you need to switch to synchronous logging (e.g. for debugging/testing purposes), you can override this behaviour by adding `-DLog4jContextSelector=org.apache.logging.log4j.core.selector.ClassLoaderContextSelector` to the node's command line or to the `jvmArgs` section of the node configuration (see [Node configuration](#)).

### 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.

### 16.5.2 SSH access

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

### 16.5.3 Database access

When running a node backed with a H2 database, the node can be configured to expose the database over a socket (see [Database access when running H2](#)).

Note that in production, exposing the database via the node is not recommended.

### 16.5.4 Monitoring your node

Like most Java servers, the node can be configured to export 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. As Java serialization in the node has been restricted for security reasons, the metrics can only be exported via a Jolokia agent.

Jolokia allows you to access the raw data and operations without connecting to the JMX port directly. Nodes can be configured to 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.
- [JMXTrans](#) 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 `jmxMonitoringHttpPort` has to be present in order 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

## Notes for production use

When using Jolokia monitoring in production, it is recommended to use a Jolokia agent that reads the metrics from the node and pushes them to the metrics storage, rather than exposing a port on the production machine/process to the internet.

Also ensure to have restrictive Jolokia access policy in place for access to production nodes. The Jolokia access is controlled via a file called `jolokia-access.xml`. Several Jolokia policy based security configuration files (`jolokia-access.xml`) are available for dev, test, and prod environments under `/config/<env>`.

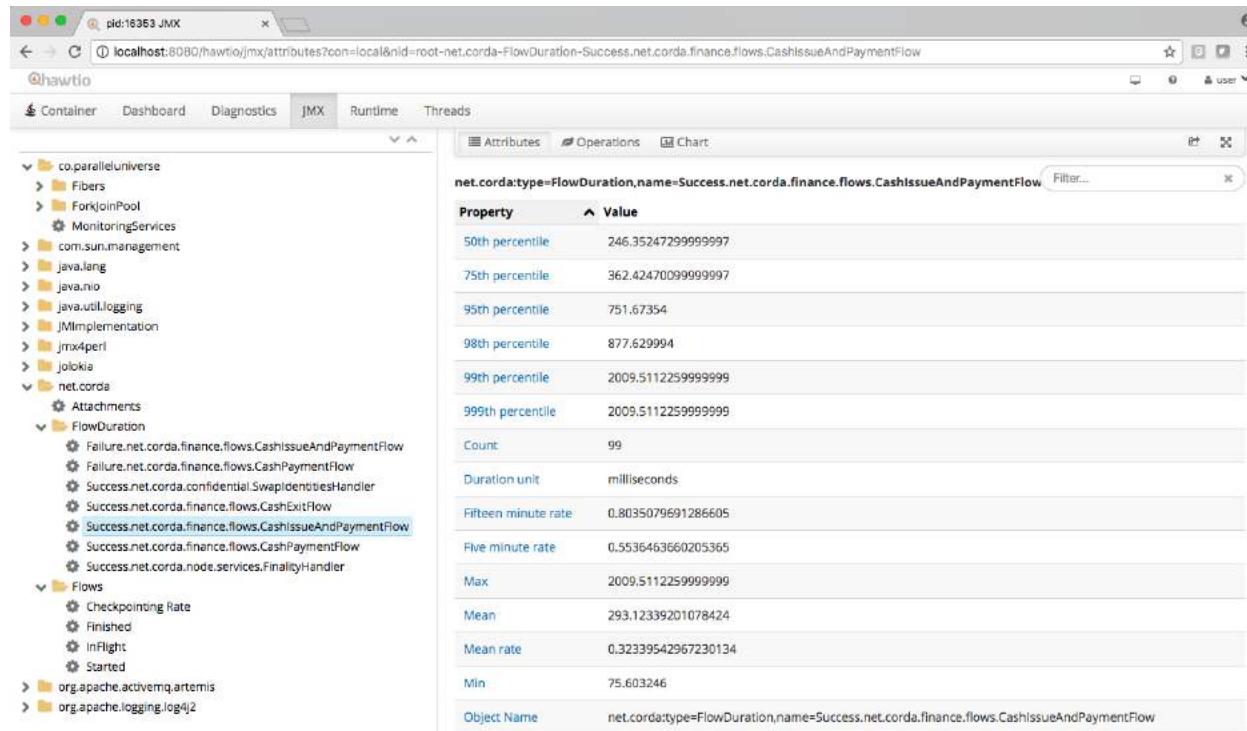
## Notes for development use

When running in dev mode, Hibernate statistics are also available via the Jolokia interface. These are disabled otherwise due to expensive run-time costs. They can be turned on and off explicitly regardless of dev mode via the `exportHibernateJMXStatistics` flag on the [database configuration](#).

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 ‘driver DSL’, 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 {}**

The following diagram illustrates Corda flow metrics visualized using hawtio:



## 16.5.5 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 512mb Java heap by default. When other overheads are added, this yields a total memory usage of about 800mb 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 -Dcapsule.jvm.args="-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.

---

## 16.5.6 Hiding sensitive data

A frequent requirement is that configuration files must not expose passwords to unauthorised readers. By leveraging environment variables, it is possible to hide passwords and other similar fields.

Take a simple node config that wishes to protect the node cryptographic stores:

```
myLegalName = "O=PasswordProtectedNode, OU=corda, L=London, C=GB"
keyStorePassword = ${KEY_PASS}
trustStorePassword = ${TRUST_PASS}
p2pAddress = "localhost:12345"
devMode = false
```

(continues on next page)

(continued from previous page)

```
networkServices {
    doormanURL = "https://cz.example.com"
    networkMapURL = "https://cz.example.com"
}
```

By delegating to a password store, and using *command substitution* it is possible to ensure that sensitive passwords never appear in plain text. The below examples are of loading Corda with the KEY\_PASS and TRUST\_PASS variables read from a program named `corporatePasswordStore`.

## Bash

```
KEY_PASS=$(corporatePasswordStore --cordaKeyStorePassword) TRUST_PASS=
↪$(corporatePasswordStore --cordaTrustStorePassword) java -jar corda.jar
```

**Warning:** If this approach is taken, the passwords will appear in the shell history.

## Windows PowerShell

```
$env:KEY_PASS=$(corporatePasswordStore --cordaKeyStorePassword); $env:TRUST_PASS=
↪$(corporatePasswordStore --cordaTrustStorePassword); java -jar corda.jar
```

For launching on Windows without PowerShell, it is not possible to perform command substitution, and so the variables must be specified manually, for example:

```
SET KEY_PASS=mypassword & SET TRUST_PASS=mypassword & java -jar corda.jar
```

**Warning:** If this approach is taken, the passwords will appear in the windows command prompt history.

### 16.5.7 Backup recommendations

Various components of the Corda platform read their configuration from the file system, and persist data to a database or into files on disk. Given that hardware can fail, operators of IT infrastructure must have a sound backup strategy in place. Whilst blockchain platforms can sometimes recover some lost data from their peers, it is rarely the case that a node can recover its full state in this way because real-world blockchain applications invariably contain private information (e.g., customer account information). Moreover, this private information must remain in sync with the ledger state. As such, we strongly recommend implementing a comprehensive backup strategy.

The following elements of a backup strategy are recommended:

#### Database replication

When properly configured, database replication prevents data loss from occurring in case the database host fails. In general, the higher the number of replicas, and the further away they are deployed in terms of regions and availability zones, the more a setup is resilient to disasters. The trade-off is that, ideally, replication should happen synchronously, meaning that a high number of replicas and a considerable network latency will impact the performance of the Corda nodes connecting to the cluster. Synchronous replication is strongly advised to prevent data loss.

## Database snapshots

Database replication is a powerful technique, but it is very sensitive to destructive SQL updates. Whether malicious or unintentional, a SQL statement might compromise data by getting propagated to all replicas. Without rolling snapshots, data loss due to such destructive updates will be irreversible. Using snapshots always implies some data loss in case of a disaster, and the trade-off is between highly frequent backups minimising such a loss, and less frequent backups consuming less resources. At present, Corda does not offer online updates with regards to transactions. Should states in the vault ever be lost, partial or total recovery might be achieved by asking third-party companies and/or notaries to provide all data relevant to the affected legal identity.

## File backups

Corda components read and write information from and to the file-system. The advice is to backup the entire root directory of the component, plus any external directories and files optionally specified in the configuration. Corda assumes the filesystem is reliable. You must ensure that it is configured to provide this assurance, which means you must configure it to synchronously replicate to your backup/DR site. If the above holds, Corda components will benefit from the following:

- Guaranteed eventual processing of acknowledged client messages, provided that the backlog of persistent queues is not lost irredeemably.
- A timely recovery from deletion or corruption of configuration files (e.g., `node.conf`, `node-info` files, etc.), database drivers, CorDapps binaries and configuration, and certificate directories, provided backups are available to restore from.

**Warning:** Private keys used to sign transactions should be preserved with the utmost care. The recommendation is to keep at least two separate copies on a storage not connected to the Internet.

## 16.6 Deploying a node to a server

### Contents

- *Deploying a node to a server*
  - *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.

---

### 16.6.1 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:**

- A supported Java distribution. The supported versions are listed in [Getting set up for CorDapp development](#)

1. As root/sys admin user - 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 /4.1/corda-4.1.jar) and place it in /opt/corda

4. (Optional) Download the [Corda webserver jar](#) (under /4.1/corda-4.1.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:

```
p2pAddress = "example.com:10002"
rpcSettings {
    address: "example.com:10003"
    adminAddress: "example.com:10004"
}
h2port = 11000
emailAddress = "you@example.com"
myLegalName = "O=Bank of Breakfast Tea, L=London, C=GB"
keyStorePassword = "cordacadevpass"
trustStorePassword = "trustpass"
devMode = false
rpcUsers= [
    {
        user=corda
        password=portal_password
        permissions=[
            ALL
        ]
    }
]
custom { jvmArgs = [ '-Xmx2048m', '-XX:+UseG1GC' ] }
```

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

- Change the p2pAddress, rpcSettings.address and rpcSettings.adminAddress values to match your server's hostname or external IP address. These are the addresses 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 (see [Node identity](#) for more details).
- If required, add RPC users

**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

```
[Unit]
Description=Corda Node - Bank of Breakfast Tea
Requires=network.target

[Service]
Type=simple
User=corda
WorkingDirectory=/opt/corda
ExecStart=/usr/bin/java -jar /opt/corda/corda.jar
Restart=on-failure

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

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

```
description "Corda Node - Bank of Breakfast Tea"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
setuid corda
chdir /opt/corda
exec java -jar /opt/corda/corda.jar
```

9. 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**
- **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

```
[Unit]
Description=Webserver for Corda Node - Bank of Breakfast Tea
Requires=network.target

[Service]
```

(continues on next page)

(continued from previous page)

```
Type=simple
User=corda
WorkingDirectory=/opt/corda
ExecStart=/usr/bin/java -jar /opt/corda/corda-webserver.jar
Restart=on-failure

[Install]
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

```
description "Webserver for Corda Node - Bank of Breakfast Tea"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
setuid corda
chdir /opt/corda
exec java -jar /opt/corda/corda-webserver.jar
```

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

12. **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.

## 16.6.2 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:

- A supported Java distribution. The supported versions are listed in [Getting set up for CorDapp development](#)
1. Create a Corda directory and download the Corda jar. Here's an example using PowerShell:

```
mkdir C:\Corda
wget http://jcenter.bintray.com/net/corda/corda/4.1/corda-4.1.jar -OutFile ..\C:\Corda\corda.jar
```

(continues on next page)

(continued from previous page)

2. 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
3. Save the below as `C:\Corda\node.conf`. See [Node configuration](#) for a description of these options:

```
p2pAddress = "example.com:10002"
rpcSettings {
    address = "example.com:10003"
    adminAddress = "example.com:10004"
}
h2port = 11000
emailAddress = "you@example.com"
myLegalName = "O=Bank of Breakfast Tea, L=London, C=GB"
keyStorePassword = "cordacadevpass"
trustStorePassword = "trustpass"
devMode = false
rpcSettings {
    useSsl = false
    standAloneBroker = false
    address = "example.com:10003"
    adminAddress = "example.com:10004"
}
custom { jvmArgs = [ '-Xmx2048m', '-XX:+UseG1GC' ] }
```

4. Make the following changes to `C:\Corda\node.conf`:
  - Change the `p2pAddress`, `rpcSettings.address` and `rpcSettings.adminAddress` values to match your server's hostname or external IP address. These are the addresses 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 (see [Node identity](#) for more details).
  - If required, add RPC users

5. Copy the required Java keystores to the node. See [Network certificates](#)
6. Download the [NSSM](#) service manager
7. Unzip `nssm-2.24\win64\nssm.exe` to `C:\Corda`
8. Save the following as `C:\Corda\nssm.bat`:

```
nssm install cordanode1 C:\ProgramData\Oracle\Java\javapath\java.exe
nssm set cordanode1 AppDirectory C:\Corda
nssm set cordanode1 AppStdout C:\Corda\service.log
nssm set cordanode1 AppStderr C:\Corda\service.log
nssm set cordanode1 Description Corda Node - Bank of Breakfast Tea
nssm set cordanode1 Start SERVICE_AUTO_START
sc start cordanode1
```

9. Modify the batch file:
  - If you are installing multiple nodes, use a different service name (`cordanode1`) for each node
  - Set an informative description

10. Provision the required certificates to your node. Contact the network permissioning service or see [Network certificates](#)
11. Run the batch file by clicking on it or from a command prompt
12. Run `services.msc` and verify that a service called `cordanode1` is present and running
13. Run `netstat -ano` and check for the ports you configured in `node.conf`
  - You may need to open the ports on the Windows firewall

### 16.6.3 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-J and Ctrl-D to exit telnet.

## 16.7 Node database

### Contents

- *Node database*
  - *Configuring the node database*
    - \* *H2*
    - \* *PostgreSQL*
    - \* *SQLServer*
  - *Node database tables*
  - *Database connection pool*

### 16.7.1 Configuring the node database

#### H2

By default, nodes store their data in an H2 database. See [Database access when running H2](#).

Nodes can also be configured to use PostgreSQL and SQL Server. However, these are experimental community contributions. The Corda continuous integration pipeline does not run unit tests or integration tests of these databases.

#### PostgreSQL

Nodes can also be configured to use PostgreSQL 9.6, using PostgreSQL JDBC Driver 42.1.4. Here is an example node configuration for PostgreSQL:

```
dataSourceProperties = {
    dataSourceClassName = "org.postgresql.ds.PGSimpleDataSource"
    dataSource.url = "jdbc:postgresql://[HOST]:[PORT]/[DATABASE]"
    dataSource.user = [USER]
    dataSource.password = [PASSWORD]
}
database = {
    transactionIsolationLevel = READ_COMMITTED
}
```

Note that:

- Database schema name can be set in JDBC URL string e.g. *currentSchema=my\_schema*
- Database schema name must either match the `dataSource.user` value to end up on the standard schema search path according to the [PostgreSQL documentation](#), or the schema search path must be set explicitly for the user.
- If your PostgreSQL database is hosting multiple schema instances (using the JDBC URL `currentSchema=my_schema`) for different Corda nodes, you will need to create a *hibernate\_sequence* sequence object manually for each subsequent schema added after the first instance. Corda doesn't provision Hibernate with a schema namespace setting and a sequence object may be not created. Run the DDL statement and replace *my\_schema* with your schema namespace:

```
CREATE SEQUENCE my_schema.hibernate_sequence INCREMENT BY 1 MINVALUE 1 MAXVALUE _9223372036854775807 START 8 CACHE 1 NO CYCLE;
```

## SQLServer

Nodes also have untested support for Microsoft SQL Server 2017, using Microsoft JDBC Driver 6.2 for SQL Server. Here is an example node configuration for SQLServer:

```
dataSourceProperties = {
    dataSourceClassName = "com.microsoft.sqlserver.jdbc.SQLServerDataSource"
    dataSource.url = "jdbc:sqlserver://[HOST];databaseName=[DATABASE_NAME]"
    dataSource.user = [USER]
    dataSource.password = [PASSWORD]
}
database = {
    transactionIsolationLevel = READ_COMMITTED
}
jarDirs = ["[FULL_PATH]/sqljdbc_6.2/enu/"]
```

Note that:

- Ensure the directory referenced by `jarDirs` contains only one JDBC driver JAR file; by default, `sqljdbc_6.2/enu` contains two JDBC JAR files for different Java versions.

### 16.7.2 Node database tables

By default, the node database has the following tables:

Table name	Columns
DATABASECHANGELOG	AUTHOR, FILENAME, DATEEXECUTED, ORDEREXECUTED, EXECTYPE, MD5SUM, DESCRIPTION, COMMENTS, TAG, LIQUIBASE, CONTEXTS, LABELS, DEPLOYMENT_ID
DATABASECHANGELOGLOCK	LOCKED, LOCKGRANTED, LOCKEDBY OGLOCK
NODE_ATTACHMENT	CONTENT, FILENAME, INSERTION_DATE, UPLOADER
NODE_ATTACHMENT_CONTRACTS	CLASS_NAME
NODE_ATTACHMENT_SIGNERS	
NODE_CHECKPOINT	CHECKPOINT_ID, CHECKPOINT_VALUE
NODE_CONTRACTSTATETRANSACTS	DETRACT_CLASS_NAME
NODE_IDENTITYHASHES	IDENTITY_HASH, IDENTITY_VALUE
NODE_INFOS	NODE_INFO_ID, NODE_INFO_HASH, PLATFORM_VERSION, SERIAL
NODE_INFO_HOSTS	HOST_NAME, PORT, NODE_INFO_ID, HOSTS_ID
NODE_INFO_PARTYCERTS	ISMAIN, OWNING_KEY_HASH, PARTY_CERT_BINARY
NODE_LINK_NODEINFOPARTY	PARTY_NAME
NODE_MESSAGE	MESSAGE_ID, INSERTION_TIME, SENDER, SEQUENCE_NUMBER
NODE_NAMEDNAME	NPKHASH
NODE_NETWORKSPARAMETERS	PARAMETERS_BYTES, SIGNATURE_BYTES, CERT, PARENT_CERT_PATH
NODE_OURKEYPAIRS	PRIVATE_KEY, PUBLIC_KEY
NODE_PROPERTY	PROPERTY_KEY, PROPERTY_VALUE
NODE_SCHEDULEDTRANSACTIONS	TRANSACTION_ID, SCHEDULED_AT
NODE_TRANSACTION	TRANSACTION_VALUE, STATE_MACHINE_RUN_ID
PK_HASH_TO_HX	EXTERNAL_ID, PUBLIC_KEY_HASH
STATE_PARTY	OUTPUT_INDEX, TRANSACTION_ID, ID, PUBLIC_KEY_HASH, X500_NAME
VAULT_FUNGIBILITYSTATESEX	TRANSACTION_ID, ISSUER_NAME, ISSUER_REF, OWNER_NAME, QUANTITY
VAULT_FUNGIBILITYSTATESPARTS	TRANSACTION_ID, PARTICIPANTS
VAULT_LINEARSTATUSINDEX	TRANSACTION_ID, EXTERNAL_ID, UUID
VAULT_LINEARSTATUSINDEXS	TRANSACTION_ID, PARTICIPANTS
VAULT_STATEOUTPUT_INDEX	TRANSACTION_ID, CONSUMED_TIMESTAMP, CONTRACT_STATE_CLASS_NAME, LOCK_ID, LOCK_TIMESTAMP, NOTARY_NAME, RECORDED_TIMESTAMP, STATE_STATUS, RELEVANCY_STATUS, CONSTRAINT_TYPE, CONSTRAINT_DATA
VAULT_TRANSACTIONNOTE	TRANSACTION_ID
V_PKEY_HASH	DEXPUBLKEY_HASH, TRANSACTION_ID, OUTPUT_INDEX, EXTERNAL_ID

### 16.7.3 Database connection pool

Corda uses [Hikari Pool](#) for creating the connection pool. To configure the connection pool any custom properties can be set in the *dataSourceProperties* section.

For example:

```
dataSourceProperties = {
    dataSourceClassName = "org.postgresql.ds.PGSimpleDataSource"
    ...
    maximumPoolSize = 10
    connectionTimeout = 50000
}
```

## 16.8 Database access when running H2

### Contents

- *Database access when running H2*
  - *Configuring the username and password*
  - *Connecting via a socket on a running node*
    - \* *Configuring the port*
    - \* *Connecting to the database*
      - *Connecting using the H2 Console*
  - *Connecting directly to the node's persistence.mv.db file*

### 16.8.1 Configuring the username and password

The database (a file called `persistence.mv.db`) is created when the node first starts up. By default, it has an administrator user `sa` and a blank password. The node requires the user with administrator permissions in order to creates tables upon the first startup or after deploying new CorDapps with their own tables. The database password is required only when the H2 database is exposed on non-localhost address (which is disabled by default).

This username and password can be changed in node configuration:

```
dataSourceProperties = {  
    dataSource.user = [USER]  
    dataSource.password = [PASSWORD]  
}
```

Note that changing the user/password for the existing node in `node.conf` will not update them in the H2 database. You need to log into the database first to create a new user or change a user's password.

### 16.8.2 Connecting via a socket on a running node

#### Configuring the port

Nodes backed by an H2 database will not expose this database by default. To configure the node to expose its internal database over a socket which can be browsed using any tool that can use JDBC drivers, you must specify the full network address (interface and port) using the `h2Settings` syntax in the node configuration.

The configuration below will restrict the H2 service to run on `localhost`:

```
h2Settings {  
    address: "localhost:12345"  
}
```

If you want H2 to auto-select a port (mimicking the old `h2Port` behaviour), you can use:

```
h2Settings {  
    address: "localhost:0"  
}
```

If remote access is required, the address can be changed to `0.0.0.0` to listen on all interfaces. A password must be set for the database user before doing so.

```
h2Settings {
    address: "0.0.0.0:12345"
}
dataSourceProperties {
    dataSource.password : "strongpassword"
}
```

---

**Note:** The previous `h2Port` syntax is now deprecated. `h2Port` will continue to work but the database will only be accessible on localhost.

---

## Connecting to the database

The JDBC URL is printed during node startup to the log and will typically look like this:

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

Any database browsing tool that supports JDBC can be used.

## Connecting using the H2 Console

- Download the [last stable 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`
- Paste the node's JDBC URL into the JDBC URL field and click `Connect`, using the default username (`sa`) and no password (unless configured otherwise)

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.

### 16.8.3 Connecting directly to the node's `persistence.mv.db` file

You can also use the H2 Console to connect directly to the node's `persistence.mv.db` file. Ensure the node is off before doing so, as access to the database file requires exclusive access. If the node is still running, the H2 Console will return the following error: `Database may be already in use: null`. Possible solutions: `close all other connection(s); use the server mode [90020-196]`.

```
jdbc:h2:~/path/to/file/persistence
```

## 16.9 Node shell

## Contents

- *Node shell*
  - *Permissions*
  - *The shell via the local terminal*
  - *The shell via SSH*
    - \* *Enabling SSH access*
    - \* *Authentication*
    - \* *Connecting to the shell*
      - *Linux and MacOS*
      - *Windows*
  - *The standalone shell*
    - \* *Starting the standalone shell*
  - *Standalone Shell via SSH*
  - *Interacting with the node via the shell*
    - \* *Shutting down the node*
    - \* *Output Formats*
    - \* *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 or standalone 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
- Shutting the node down.

### 16.9.1 Permissions

When accessing the shell (embedded, standalone, via SSH) RPC permissions are required. This is because the shell actually communicates with the node using RPC calls.

- Watching flows (`flow watch`) requires `InvokeRpc.stateMachinesFeed`.
- Starting flows requires `InvokeRpc.startTrackedFlowDynamic`, `InvokeRpc.registeredFlows` and `InvokeRpc.wellKnownPartyFromX500Name`, as well as a permission for the flow being started.
- Killing flows (`flow kill`) requires `InvokeRpc.killFlow`. This currently allows the user to kill *any* flow, so please be careful when granting it!

### 16.9.2 The shell via the local terminal

---

**Note:** Local terminal shell works only in development mode!

---

The shell will display in the node’s terminal window. It connects to the node as ‘shell’ user with password ‘shell’ (which is only available in dev mode). It may be disabled by passing the `--no-local-shell` flag when running the node.

### 16.9.3 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. 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.

---

**Note:** 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. 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.

## 16.9.4 The standalone shell

The standalone shell is a standalone application interacting with a Corda node via RPC calls. RPC node permissions are necessary for authentication and authorisation. Certain operations, such as starting flows, require access to CordApps jars.

### Starting the standalone shell

Run the following command from the terminal:

```
corda-shell [-hvV] [--logging-level=<loggingLevel>] [--password=<password>]
[--sshd-hostkey-directory=<sshdHostKeyDirectory>]
[--sshd-port=<sshdPort>] [--truststore-file=<trustStoreFile>]
[--truststore-password=<trustStorePassword>]
[--truststore-type=<trustStoreType>] [--user=<user>] [-a=<host>]
[-c=<cordappDirectory>] [-f=<configFile>] [-o=<commandsDirectory>]
[-p=<port>] [COMMAND]
```

Where:

- `--config-file=<configFile>, --f` The path to the shell configuration file, used instead of providing the rest of the command line options.
- `--cordapp-directory=<cordappDirectory>, -c` The path to the directory containing CorDApp jars, CorDApps are required when starting flows.
- `--commands-directory=<commandsDirectory>, -o` The path to the directory containing additional CRaSH shell commands.
- `--host, -a`: The host address of the Corda node.
- `--port, -p`: The RPC port of the Corda node.
- `--user=<user>`: The RPC user name.
- `--password=<password>` The RPC user password. If not provided it will be prompted for on startup.
- `--sshd-port=<sshdPort>` Enables SSH server for shell.
- `--sshd-hostkey-directory=<sshHostKeyDirectory>`: The directory containing the hostkey.pem file for the SSH server.
- `--truststore-password=<trustStorePassword>`: The password to unlock the TrustStore file.
- `--truststore-file=<trustStoreFile>`: The path to the TrustStore file.
- `--truststore-type=<trustStoreType>`: The type of the TrustStore (e.g. JKS).
- `--verbose, --log-to-console, -v`: If set, prints logging to the console as well as to a file.
- `--logging-level=<loggingLevel>`: Enable logging at this level and higher. Possible values: ERROR, WARN, INFO, DEBUG, TRACE. Default: INFO.
- `--help, -h`: Show this help message and exit.
- `--version, -V`: Print version information and exit.

Additionally, the `install-shell-extensions` subcommand can be used to install the `corda-shell` alias and auto completion for bash and zsh. See [Shell extensions for CLI Applications](#) for more info.

The format of config-file:

```
node {
    addresses {
        rpc {
            host : "localhost"
            port : 10006
        }
    }
}
shell {
    workDir : /path/to/dir
}
extensions {
    cordapps {
        path : /path/to/cordapps/dir
    }
    sshd {
        enabled : "false"
        port : 2223
    }
}
```

(continues on next page)

(continued from previous page)

```
ssl {
    keystore {
        path: "/path/to/keystore"
        type: "JKS"
        password: password
    }
    trustore {
        path: "/path/to/trusttore"
        type: "JKS"
        password: password
    }
}
user : demo
password : demo
```

## 16.9.5 Standalone Shell via SSH

The standalone shell can embed an SSH server which redirects interactions via RPC calls to the Corda node. To run SSH server use `--sshd-port` option when starting standalone shell or `extensions.sshd` entry in the configuration file. For connection to SSH refer to [Connecting to the shell](#). Certain operations (like starting Flows) will require Shell's `--cordpass-directory` to be configured correctly (see [Starting the standalone shell](#)).

## 16.9.6 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](#).

### Shutting down the node

You can shut the node down via shell:

- `gracefulShutdown` will put node into draining mode, and shut down when there are no flows running
- `shutdown` will shut the node down immediately

### Output Formats

You can choose the format in which the output of the commands will be shown.

To see what is the format that's currently used, you can type `output-format get`.

To update the format, you can type `output-format set json`.

The currently supported formats are `json`, `yaml`. The default format is `yaml`.

## 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
- `flow kill` kills a single flow, as identified by its UUID.

## 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`)
- Nested classes are referenced using `$`. For example, the `net.corda.finance.contracts.asset.Cash.State` class is referenced as `net.corda.finance.contracts.asset.Cash$State` (note the `$`)

---

**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=Controller, 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:

```
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.

### 16.9.7 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!

### 16.9.8 Limitations

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

- Flows cannot be run unless they override the progress tracker
- If a command requires an argument of an abstract type, the command cannot be run because the concrete subclass to use cannot be specified using the YAML syntax
- 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

## 16.10 Interacting with a node

### Contents

- *Interacting with a node*
  - *Overview*
  - *Connecting to a node via RPC*
  - *RPC permissions*
    - \* *Granting flow permissions*
    - \* *Granting other RPC permissions*
    - \* *Granting all permissions*
  - *RPC security management*
    - \* *Authentication/authorisation data*
    - \* *Password encryption*

- \* *Caching user accounts data*
- *Observables*
- *Futures*
- *Versioning*
- *Thread safety*
- *Error handling*
- *Reconnecting RPC clients*
- *Wire security*
- *Whitelisting classes with the Corda node*

### 16.10.1 Overview

To interact with your node, you need to write a client in a JVM-compatible language using the `CordaRPCClient` class. This class allows you to connect to your node via a message queue protocol and provides a simple RPC interface for interacting with the node. You make calls on a JVM object as normal, and the marshalling back-and-forth is handled for you.

**Warning:** The built-in Corda webserver is deprecated and unsuitable for production use. If you want to interact with your node via HTTP, you will need to stand up your own webserver that connects to your node using the `CordaRPCClient` class. You can find an example of how to do this using the popular Spring Boot server [here](#).

### 16.10.2 Connecting to a node via RPC

To use `CordaRPCClient`, you must add `net.corda:corda-rpc:$corda_release_version` as a `cordaCompile` dependency in your client's `build.gradle` file.

`CordaRPCClient` has a `start` method that takes the node's RPC address and returns a `CordaRPCConnection`. `CordaRPCConnection` has a `proxy` method that takes an RPC username and password and returns a `CordaRPCOps` object that you can use to interact with the node.

Here is an example of using `CordaRPCClient` to connect to a node and log the current time on its internal clock:

```
import net.corda.client.rpc.CordaRPCClient
import net.corda.core.utilities.NetworkHostAndPort.Companion.parse
import net.corda.core.utilities.loggerFor
import org.slf4j.Logger

class ClientRpcExample {
    companion object {
        val logger: Logger = loggerFor<ClientRpcExample>()
    }

    fun main(args: Array<String>) {
        require(args.size == 3) { "Usage: TemplateClient <node address> <username>\n" +
            "  <password>" }
        val nodeAddress = parse(args[0])
        val username = args[1]
```

(continues on next page)

(continued from previous page)

```

    val password = args[2]

    val client = CordaRPCCClient(nodeAddress)
    val connection = client.start(username, password)
    val cordaRPCOperations = connection.proxy

    logger.info(cordaRPCOperations.currentTimeMillis().toString())

    connection.notifyServerAndClose()
}
}

import net.corda.client.rpc.CordaRPCClient;
import net.corda.client.rpc.CordaRPCCConnection;
import net.corda.core.messaging.CordaRPCOps;
import net.corda.core.utilities.NetworkHostAndPort;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class ClientRpcExample {
    private static final Logger logger = LoggerFactory.getLogger(ClientRpcExample.
    <class>);

    public static void main(String[] args) {
        if (args.length != 3) {
            throw new IllegalArgumentException("Usage: TemplateClient <node address>
    <username> <password>");
        }
        final NetworkHostAndPort nodeAddress = NetworkHostAndPort.parse(args[0]);
        String username = args[1];
        String password = args[2];

        final CordaRPCCClient client = new CordaRPCCClient(nodeAddress);
        final CordaRPCCConnection connection = client.start(username, password);
        final CordaRPCOps cordaRPCOperations = connection.getProxy();

        logger.info(cordaRPCOperations.currentTimeMillis().toString());

        connection.notifyServerAndClose();
    }
}

```

**Warning:** The returned `CordaRPCCConnection` 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 `CordaRPCCClient` 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 further information on using the RPC API, see [Using the client RPC API](#).

### 16.10.3 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. Permissions are not required to interact with the node via the shell, unless the shell is being accessed via SSH.

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  
    }]
```

(continues on next page)

(continued from previous page)

```

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"  

        ]  

    },  

    ...
]

```

#### 16.10.4 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>"  

            }  

        }  

    }  

}

```

(continues on next page)

(continued from previous page)

```
        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.

## **Authentication/authorisation data**

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.

**DB** 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.

**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.

---

## Password encryption

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](#).

## Caching user accounts data

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.

### 16.10.5 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 will be disconnected. 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. For Kotlin users there is a convenience extension method called `notUsed()` which can be called on an observable to automate this step.

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. If you set `-Dnet.corda.client.rpc.trackRpcCallSites=true` on the JVM

command line then this warning comes with a stack trace showing where the RPC that returned the forgotten observable was called from. This feature is off by default because tracking RPC call sites is moderately slow.

**Note:** Observables can only be used as return arguments of an RPC call. It is not currently possible to pass Observables as parameters to the RPC methods. In other words the streaming is always server to client and not the other way around.

### 16.10.6 Futures

A method can also return a `CordaFuture` 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.

### 16.10.7 Versioning

The client RPC protocol is versioned using the node's platform version number (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).

The RPC client library defaults to requiring the platform version it was built with. That means if you use the client library released as part of Corda N, then the node it connects to must be of version N or above. This is checked when the client first connects. If you want to override this behaviour, you can alter the `minimumServerProtocolVersion` field in the `CordaRPCClientConfiguration` object passed to the client. Alternatively, just link your app against an older version of the library.

### 16.10.8 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.

### 16.10.9 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 the behaviour depends on the `devMode` node configuration option.

In `devMode`, 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.

When not in `devMode`, the server will mask exceptions not meant for clients and return an `InternalNodeException` instead. This does not expose internal information to clients, strengthening privacy and security. CorDapps can have exceptions implement `ClientRelevantError` to allow them to reach RPC clients.

### 16.10.10 Reconnecting RPC clients

In the current version of Corda the RPC connection and all the observables that are created by a client will just throw exceptions and die when the node or TCP connection become unavailable.

It is the client's responsibility to handle these errors and reconnect once the node is running again. Running RPC commands against a stopped node will just throw exceptions. Previously created Observables will not emit any events after the node restarts. The client must explicitly re-run the command and re-subscribe to receive more events.

RPCs which have a side effect, such as starting flows, may have executed on the node even if the return value is not received by the client. The only way to confirm is to perform a business-level query and retry accordingly. The sample *runFlowWithLogicalRetry* helps with this.

In case users require such a functionality to write a resilient RPC client we have a sample that showcases how this can be implemented and also a thorough test that demonstrates it works as expected.

The code that performs the reconnecting logic is: [ReconnectingCordaRPCOps.kt](#).

---

**Note:** This sample code is not exposed as an official Corda API, and must be included directly in the client codebase and adjusted.

---

The usage is showcased in the: [RpcReconnectTests.kt](#). In case resiliency is a requirement, then it is recommended that users will write a similar test.

How to initialize the *ReconnectingCordaRPCOps*:

```
val bankAReconnectingRpc = ReconnectingCordaRPCOps(bankAAddress, demoUser.  
→username, demoUser.password)
```

How to track the vault :

```
val vaultFeed = bankAReconnectingRpc.vaultTrackWithPagingSpec(  
    Cash.State::class.java,  
    QueryCriteria.VaultQueryCriteria(),  
    PageSpecification(1, 1)  
    val vaultObserverHandle = vaultFeed.updates.asReconnecting().subscribe {  
        →update: Vault.Update<Cash.State> ->  
            log.info("vault update produced ${update.produced.map { it.state.data.  
        →amount }} consumed ${update.consumed.map { it.ref }}")  
            vaultEvents.add(update)  
    }
```

How to start a flow with a logical retry function that checks for the side effects of the flow:

```
bankAReconnectingRpc.runFlowWithLogicalRetry(  
    runFlow = { rpc ->  
        log.info("Starting CashIssueAndPaymentFlow for $amount")  
        val flowHandle = rpc.startTrackedFlowDynamic(  
            CashIssueAndPaymentFlow::class.java,  
            baseAmount.plus(Amount.parseCurrency("$amount USD  
        →")),  
            issuerRef,  
            bankB.nodeInfo.legalIdentities.first(),  
            false,  
            notary  
        )  
        val flowId = flowHandle.id
```

(continues on next page)

(continued from previous page)

```

        log.info("Started flow $amount with flowId: $flowId")
        flowProgressEvents.addEvent(flowId, null)

        // No reconnecting possible.
        flowHandle.progress.subscribe(
            { prog ->
                flowProgressEvents.addEvent(flowId, prog)
                log.info("Progress $flowId : $prog")
            },
            { error ->
                log.error("Error thrown in the flow progress",
                    "observer", error)
            })
        flowHandle.id
    },
    hasFlowStarted = { rpc ->
        // Query for a state that is the result of this flow.
        val criteria = QueryCriteria.
        ↪VaultCustomQueryCriteria(builder { CashSchemaV1.PersistentCashState::pennies.
        ↪equal(amount.toLong() * 100) }, status = Vault.StateStatus.ALL)
        val results = rpc.vaultQueryByCriteria(criteria, Cash.
        ↪State::class.java)
        log.info("$amount - Found states ${results.states}")
        // The flow has completed if a state is found
        results.states.isNotEmpty()
    },
    onFlowConfirmed = {
        flowsCountdownLatch.countDown()
        log.info("Flow started for $amount. Remaining flows: ${flowsCountdownLatch.count}")
    }
)

```

Note that, as shown by the test, during reconnecting some events might be lost.

```

        // Check that enough vault events were received.
        // This check is fuzzy because events can go missing during node restarts.
        // Ideally there should be nrOfFlowsToRun events receive but some might
        ↪get lost for each restart.
        assertTrue(vaultEvents!!.size + nrFailures * 2 >= nrOfFlowsToRun, "Not
        ↪all vault events were received")

```

## 16.10.11 Wire security

If TLS communications to the RPC endpoint are required the node should be configured with `rpcSettings.useSSL=true` see [Node configuration](#). The node admin should then create a node specific RPC certificate and key, by running the node once with `generate-rpc-ssl-settings` command specified (see [Node command-line options](#)). The generated RPC TLS trust root certificate will be exported to a `certificates/export/rpcssltruststore.jks` file which should be distributed to the authorised RPC clients.

The connecting `CordaRPCClient` code must then use one of the constructors with a parameter of type `ClientRpcSslOptions` ([JavaDoc](#)) and set this constructor argument with the appropriate path for the `rpcssltruststore.jks` file. The client connection will then use this to validate the RPC server handshake.

Note that RPC TLS does not use mutual authentication, and delegates fine grained user authentication and authorisation to the RPC security features detailed above.

### 16.10.12 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](#).

## 16.11 Creating nodes locally

### Contents

- *Creating nodes locally*
  - *Handcrafting a node*
  - *The Cordform task*
    - \* *Signing Cordapp JARs*
    - \* *Specifying a custom webserver*
  - *The Dockerform task*
  - *Running the Cordform/Dockerform tasks*

### 16.11.1 Handcrafting a node

A node can be created manually by creating a folder that contains the following items:

- The Corda JAR
  - Can be downloaded from <https://r3.bintray.com/corda/net/corda/corda/> (under /4.1/corda-4.1.jar)
- A node configuration file entitled `node.conf`, configured as per [Node configuration](#)
- A folder entitled `cordapps` containing any CorDapp JARs you want the node to load
- **Optional:** A webserver JAR entitled `corda-webserver.jar` that will connect to the node via RPC
  - The (deprecated) default webserver can be downloaded from <http://r3.bintray.com/corda/net/corda/corda-webserver/> (under /4.1/corda-4.1.jar)
  - A Spring Boot alternative can be found here: <https://github.com/corda/spring-webserver>

The remaining files and folders described in [Node folder structure](#) will be generated at runtime.

### 16.11.2 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"

    nodeDefaults {
```

(continues on next page)

(continued from previous page)

```

cordapps = [
    "net.corda:corda-finance-contracts:$corda_release_version",
    "net.corda:corda-finance-workflows:$corda_release_version",
    "net.corda:corda-confidential-identities:$corda_release_version"
]
}

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
    // Starts an internal SSH server providing a management shell on the node.
    sshdPort 2223
    extraConfig = [
        // Setting the JMX reporter type.
        jmxReporterType: 'JOLOKIA',
        // Setting the H2 address.
        h2Settings: [ address: 'localhost:10030' ]
    ]
}
node {
    name "O=PartyA,L=London,C=GB"
    p2pPort 10005
    rpcSettings {
        port 10006
        adminPort 10026
    }
    webPort 10007
    h2Port 10008
    // 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
    // Grants user1 the ability to start the MyFlow flow.
    rpcUsers = [[ user: "user1", "password": "test", "permissions": ["StartFlow.
    ↵net.corda.flows.MyFlow"] ]]
}
}

```

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 `cordafinance` CorDapp
- PartyA and PartyB nodes that:
    - Are not offering any services
    - Will have a webserver (since `webPort` is defined)
    - Are running the `cordafinance` 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!

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. Note, if you add a property already created by the '`deployNodes`' task, both properties will be present in the file. 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 `node-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/node-b.conf"
    }
}
```

Cordform parameter `drivers` of the `node` entry lists paths of the files to be copied to the `./drivers` subdirectory of the node. To copy the same file to all nodes `ext.drivers` can be defined in the top level and reused for each node via `drivers=ext.drivers`.

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    ext.drivers = ['lib/my_common_jar.jar']
    [...]
    node {
        name "O=PartyB,L>New York,C=US"
        [...]
        drivers = ext.drivers + ['lib/my_specific_jar.jar']
    }
}
```

## Signing Cordapp JARs

The default behaviour of Cordform is to deploy CorDapp JARs “as built”:

- prior to Corda 4 all CorDapp JARs were unsigned.
- as of Corda 4, CorDapp JARs created by the Gradle *cordapp* plugin are signed by a Corda development certificate by default.

The Cordform `signing` entry can be used to override and customise the signing of CorDapp JARs. Signing the CorDapp enables its contract classes to use signature constraints instead of other types of the constraints [API: Contract Constraints](#).

The sign task may use an external keystore, or create a new one. The `signing` entry may contain the following parameters:

- `enabled` the control flag to enable signing process, by default is set to `false`, set to `true` to enable signing
- `all` if set to `true` (by default) all CorDapps inside `cordapp` subdirectory will be signed, otherwise if `false` then only the generated Cordapp will be signed
- `options` any relevant parameters of [SignJar ANT task](#) and [GenKey ANT task](#), by default the JAR file is signed by Corda development key, the external keystore can be specified, the minimal list of required options is shown below, for other options referer to [SignJar task](#):
  - `keystore` the path to the keystore file, by default `cordadevkeys.jks` keystore is shipped with the plugin
  - `alias` the alias to sign under, the default value is `cordaintermediateca`
  - `storepass` the keystore password, the default value is `cordacadevpass`
  - `keypass` the private key password if it’s different than the password for the keystore, the default value is `cordacadevkeypass`
  - `storetype` the keystore type, the default value is `JKS`
  - `dname` the distinguished name for entity, the option is used when `generateKeystore true` only
  - `keyalg` the method to use when generating name-value pair, the value defaults to `RSA` as Corda doesn’t support `DSA`, the option is used when `generateKeystore true` only
- `generateKeystore` the flag to generate a keystore, it is set to `false` by default. If set to `true` then ad hock keystore is created and its key is used instead of the default Corda development key or any external key. The same options to specify an external keystore are used to define the newly created keystore. Additionally `dname` and `keyalg` are required. Other options are described in [GenKey task](#). If the existing keystore is already present the task will reuse it, however if the file is inside the `build` directory, then it will be deleted when Gradle `clean` task is run.

The example below shows the minimal set of options needed to create a dummy keystore:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    signing {
        enabled true
        generateKeystore true
        all false
        options {
            keystore "./build/nodes/jarSignKeystore.p12"
            alias "cordapp-signer"
            storepass "secret1!"
            storetype "PKCS12"
            dname "OU=Dummy Cordapp Distributor, O=Corda, L=London, C=GB"
            keyalg "RSA"
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

//...

```

Contracts classes from signed CorDapp JARs will be checked by signature constraints by default. You can force them to be checked by zone constraints by adding contract class names to `includeWhitelist` entry, the list will generate `include_whitelist.txt` file used internally by [Network Bootstrapper](#) tool. Refer to [API: Contract Constraints](#) to understand implication of different constraint types before adding `includeWhitelist` to `deployNodes` task. The snippet below configures contracts classes from Finance CorDapp to be verified using zone constraints instead of signature constraints:

```

task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    includeWhitelist = [ "net.corda.finance.contracts.asset.Cash", "net.corda.finance.
    ↳contracts.asset.CommercialPaper" ]
}
//...

```

### Specifying a custom webserver

By default, any node listing a web port will use the default development webserver, which is not production-ready. You can use your own webserver JAR instead by using the `webserverJar` argument in a `Cordform` node configuration block:

```

node {
    name "O=PartyA, L>New York, C=US"
    webPort 10005
    webserverJar "lib/my_webserver.jar"
}

```

The webserver JAR will be copied into the node's `build` folder with the name `corda-webserver.jar`.

**Warning:** This is an experimental feature. There is currently no support for reading the webserver's port from the node's `node.conf` file.

### 16.11.3 The Dockerform task

The `Dockerform` is a sister task of `Cordform` that provides an extra file allowing you to easily spin up nodes using `docker-compose`. It supports the following configuration options for each node:

- `name`
- `notary`
- `cordapps`
- `rpcUsers`
- `useTestClock`

There is no need to specify the nodes' ports, as every node has a separate container, so no ports conflict will occur. Every node will expose port 10003 for RPC connections.

The nodes' webservers will not be started. Instead, you should interact with each node via its shell over SSH (see the [node configuration options](#)). You have to enable the shell by adding the following line to each node's `node.conf` file:

```
sshd { port = 2222 }
```

Where 2222 is the port you want to open to SSH into the shell.

Below you can find the example task from the [IRS Demo](#) 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 deployNodes(type: net.corda.plugins.Dockerform, dependsOn: ['jar']) {

    nodeDefaults {
        cordapps = [
            "net.corda:corda-finance-contracts:$corda_release_version",
            "net.corda:corda-finance-workflows:$corda_release_version",
            "net.corda:corda-confidential-identities:$corda_release_version"
        ]
    }

    node {
        name "O=Notary Service,L=Zurich,C=CH"
        notary = [validating : true]
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Bank A,L=London,C=GB"
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Bank B,L=New York,C=US"
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Regulator,L=Moscow,C=RU"
        rpcUsers = rpcUsersList
        useTestClock true
    }
}
```

### 16.11.4 Running the Cordform/Dockerform tasks

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.

If the task is a Dockerform task, running the task will also create an additional `Dockerfile` in each node directory, and a `docker-compose.yml` file in the `build/nodes` directory.

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

## 16.12 Running nodes locally

### Contents

- *Running nodes locally*
  - *Starting a Corda node using DemoBench*
  - *Starting a Corda node from the command line*
    - \* *Setting JVM arguments*
  - *Starting all nodes at once on a local machine from the command line*
    - \* *Native*
    - \* *docker-compose*
  - *Starting all nodes at once on a remote machine from the command line*

---

**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.

### 16.12.1 Starting a Corda node using DemoBench

See the instructions in [DemoBench](#).

### 16.12.2 Starting a Corda node from the command line

Run a 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`).

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.

## Setting JVM arguments

There are several ways of setting JVM arguments for the node process (particularly the garbage collector and the memory settings). They are listed here in order of increasing priority, i.e. if the same flag is set in a way later in this list, it will override anything set earlier.

**Default arguments in capsule** The capsules corda node has default flags set to `-Xmx512m -XX:+UseG1GC` - this gives the node (a relatively low) 512 MB of heap space and turns on the G1 garbage collector, ensuring low pause times for garbage collection.

**Node configuration** The node configuration file can specify custom default JVM arguments by adding a section like:

```
custom = {
    jvmArgs: [ '-Xmx1G', '-XX:+UseG1GC' ]
}
```

Note that this will completely replace any defaults set by capsule above, not just the flags that are set here, so if you use this to set e.g. the memory, you also need to set the garbage collector, or it will revert to whatever default your JVM is using.

**Capsule specific system property** You can use a special system property that Capsule understands to set JVM arguments only for the Corda process, not the launcher that actually starts it:

```
java -Dcapsule.jvm.args="-Xmx:1G" corda.jar
```

Setting a property like this will override any value for this property, but not interfere with any other JVM arguments that are configured in any way mentioned above. In this example, it would reset the maximum heap memory to `-Xmx1G` but not touch the garbage collector settings. This is particularly useful for either setting large memory allowances that you don't want to give to the launcher or for setting values that can only be set on one process at a time, e.g. a debug port.

**Command line flag** You can set JVM args on the command line that apply to the launcher process and the node process as in the example above. This will override any value for the same flag set any other way, but will leave any other JVM arguments alone.

### 16.12.3 Starting all nodes at once on a local machine from the command line

#### Native

If you created your nodes using `deployNodes`, a `runnodes` shell script (or batch file on Windows) will have been generated to allow you to quickly start up all nodes and their webservers. `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`

**Warning:** On macOS, do not click/change focus until all the node terminal windows have opened, or some processes may fail to start.

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 a Corda node from the command line](#).

### docker-compose

If you created your nodes using `Dockerform`, the `docker-compose.yml` file and corresponding `Dockerfile` for nodes has been created and configured appropriately. Navigate to `build/nodes` directory and run `docker-compose up` command. This will startup nodes inside new, internal network. After the nodes are started up, you can use `docker ps` command to see how the ports are mapped.

**Warning:** You need both Docker and `docker-compose` installed and enabled to use this method. Docker CE (Community Edition) is enough. Please refer to [Docker CE documentation](#) and [Docker Compose documentation](#) for installation instructions for all major operating systems.

#### 16.12.4 Starting all nodes at once on a remote machine from the command line

By default, `Cordform` expects the nodes it generates to be run on the same machine where they were generated. In order to run the nodes remotely, the nodes can be deployed locally and then copied to a remote server. If after copying the nodes to the remote machine you encounter errors related to `localhost` resolution, you will additionally need to follow the steps below.

To create nodes locally and run on a remote machine perform the following steps:

1. Configure `Cordform` task and deploy the nodes locally as described in [Creating nodes locally](#).
2. Copy the generated directory structure to a remote machine using e.g. Secure Copy.
3. Optionally, bootstrap the network on the remote machine.

This is optional step when a remote machine doesn't accept `localhost` addresses, or the generated nodes are configured to run on another host's IP address.

If required change host addresses in top level configuration files `[NODE NAME]_node.conf` for entries `p2pAddress`, `rpcSettings.address` and `rpcSettings.adminAddress`.

Run the network bootstrapper tool to regenerate the nodes network map (see for more explanation [Network Bootstrapper](#)):

```
java -jar corda-tools-network-bootstrapper-Master.jar --dir  
<nodes-root-dir>
```

4. Run nodes on the remote machine using `runnodes command`.

The above steps create a test deployment as `deployNodes` Gradle task would do on a local machine.

## 17.1 What is a compatibility zone?

Every Corda node is part of a “zone” (also sometimes called a Corda network) that is *permissioned*. Production deployments require a secure certificate authority. We use the term “zone” to refer to a set of technically compatible nodes reachable over a TCP/IP network like the internet. The word “network” is used in Corda but can be ambiguous with the concept of a “business network”, which is usually more like a membership list or subset of nodes in a zone that have agreed to trade with each other.

### 17.1.1 How do I become part of a compatibility zone?

#### Bootstrapping a compatibility zone

You can easily bootstrap a compatibility zone for testing or pre-production use with either the [Network Bootstrapper](#) or the [Corda Network Builder](#) tools.

#### Joining an existing compatibility zone

After the testing and pre-production phases, users are encouraged to join an existing compatibility zone such as Corda Network (the main compatibility zone) or the Corda Testnet. See [Joining an existing compatibility zone](#).

#### Setting up a dynamic compatibility zone

Some users may also be interested in setting up their own dynamic compatibility zone. For instructions and a discussion of whether this approach is suitable for you, see [Setting up a dynamic compatibility zone](#).

## 17.2 Network certificates

### Contents

- *Network certificates*
  - *Certificate hierarchy*
  - *Key pair and certificate formats*
  - *Certificate role extension*

### 17.2.1 Certificate hierarchy

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

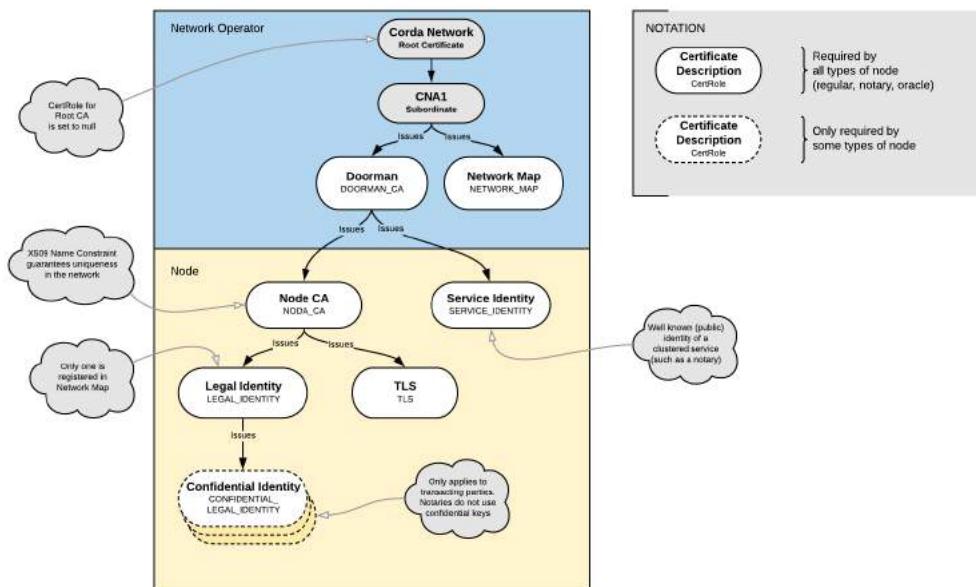
- The **root network CA** that defines the extent of a compatibility zone
- The **doorman CA** that is used instead of the root network CA for day-to-day key signing to reduce the risk of the root network CA's private key being compromised. This is equivalent to an intermediate certificate in the web PKI
- Each node also serves as its own CA, issuing the child certificates that it uses to sign its identity keys and TLS certificates

Each certificate contains an X.509 extension that defines the certificate/key's role in the system (see below for details). It also uses X.509 name constraints to ensure that the X.500 names that encode human meaningful identities are propagated to all the child certificates properly. The following constraints are imposed:

- Doorman certificates are issued by a network root. Network root certs do not contain a role extension
- Node certificates are signed by a doorman certificate (as defined by the extension)
- Legal identity/TLS certificates are issued by a certificate marked as node CA
- Confidential 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 the 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.

We can visualise the permissioning structure as follows:



## 17.2.2 Key pair and certificate formats

The required key pairs and certificates take the form of the following Java-style keystores (this may change in future to support PKCS#12 keystores) in the node's <workspace>/certificates/ folder:

- network-root-truststore.jks, the network/zone operator's root certificate as provided by them with a standard password. Can be deleted after initial registration
- truststore.jks, the network/zone operator's root certificate in keystore with a locally configurable password as protection against certain attacks
- nodekeystore.jks, which stores the node's identity key pairs and certificates
- sslkeystore.jks, which stores the node's TLS key pair and certificate

The key pairs and certificates must obey the following restrictions:

1. The certificates must follow the [X.509v3 standard](#)
2. The TLS certificates must follow the [TLS v1.2 standard](#)
3. 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)
  - ECDSA using the Koblitz k1 curve (secp256k1)
  - RSA with 3072-bit key size or higher
4. The node CA certificates must have the basic constraints extension set to true
5. The TLS certificates must have the basic constraints extension set to false

## 17.2.3 Certificate role extension

Corda certificates have a custom X.509v3 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 need not be aware of these. However, if node certificates are to be managed by external tools, such as those provided as part of an existing PKI solution deployed within an organisation, it is important to recognise these extensions and the constraints noted above.

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

## 17.3 The network map

### Contents

- *The network map*
  - *HTTP network map protocol*
  - *The additional-node-infos directory*
  - *Network parameters*
  - *Network parameters update process*
    - \* *Auto Acceptance*
    - \* *Manual Acceptance*
  - *Cleaning the network map cache*

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).

---

### 17.3.1 HTTP network map protocol

If the node is configured with the `compatibilityZoneURL` config then it first uploads its own signed `NodeInfo` to the server at that URL (and each time it changes on startup) and then proceeds to download the entire network map from the same server. 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 public network map object. The entire object is signed with the network map certificate which is also attached.
GET	/network-map/{uuid}	Retrieve the current signed private network map object with given uuid. Format is the same as for /network-map endpoint.
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.
GET	/network-map/my-hostname	Retrieve the IP address of the caller (and <b>not</b> of the network map).

Network maps hosted by R3 or other parties using R3's commercial network management tools typically also provide the following endpoints as a convenience to operators and other users

---

**Note:** we include them here as they can aid debugging but, for the avoidance of doubt, they are not a formal part of the spec and the node will operate even in their absence.

---

Request method	Path	Description
GET	/network-map/json	Retrieve the current public network map formatted as a JSON document.
GET	/network-map/json/{uuid}	Retrieve the current network map for a private network indicated by the <code>uuid</code> parameter formatted as a JSON document.
GET	/network-map/json/node-infos	Retrieve a human readable list of the currently registered <code>NodeInfo</code> files in the public network formatted as a JSON document.
GET	/network-map/json/node-infos/{uid}	Retrieve a human readable list of the currently registered <code>NodeInfo</code> files in the specified private network map.
GET	/network-map/json/node-info/{hash}	Retrieve a human readable version of a <code>NodeInfo</code> formatted as a JSON document.

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.

### 17.3.2 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 `generate-node-info` sub-command 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 [Network Bootstrapper](#).

### 17.3.3 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** Maximum allowed size in bytes of an individual message sent over the wire. Note that attachments are a special case and may be fragmented for streaming transfer, however, an individual transaction or flow message may not be larger than this value.

**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 SHA-256 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.

**packageOwnership** List of the network-wide java packages that were successfully claimed by their owners. Any CorDapp JAR that offers contracts and states in any of these packages must be signed by the owner. This ensures that when a node encounters an owned contract it can uniquely identify it and knows that all other nodes can do the same. Encountering an owned contract in a JAR that is not signed by the rightful owner is most likely a sign of malicious behaviour, and should be reported. The transaction verification logic will throw an exception when this happens. Read more about *Package ownership* here [design/data-model-upgrades/package-namespace-ownership](#).

More parameters will be added in future releases to regulate things like allowed port numbers, whether or not IPv6 connectivity is required for zone members, required cryptographic algorithms and roll-out schedules (e.g. for moving to post quantum cryptography), parameters related to SGX and so on.

#### 17.3.4 Network parameters update process

Network parameters are controlled by the zone operator of the Corda network that you are a member of. Occasionally, they may need to change these parameters. There are many reasons that can 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 roll-out of network parameter changes.

---

Updating of the parameters by the zone operator is done in two phases: 1. Advertise the proposed network parameter update to the entire network. 2. Switching the network onto the new parameters - also known as a *flag day*.

The proposed parameter update will include, along with the new parameters, a human-readable description of the changes as well as the deadline for accepting the update. The acceptance deadline marks the date and time that the zone operator intends to switch the entire network onto the new parameters. This will be a reasonable amount of time in the future, giving the node operators time to inspect, discuss and accept the 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
```

(continues on next page)

(continued from previous page)

```

 * @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
)

```

## Auto Acceptance

If the only changes between the current and new parameters are for auto-acceptable parameters then, unless configured otherwise, the new parameters will be accepted without user input. The following parameters with the `@AutoAcceptable` annotation are auto-acceptable:

```

/**
 * 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.
 *
 * @property minimumPlatformVersion Minimum version of Corda platform that is
 ↪required for nodes in the network.
 * @property notaries List of well known and trusted notary identities with
 ↪information on validation type.
 * @property maxMessageSize This is currently ignored. However, it will be wired up
 ↪in a future release.
 * @property maxTransactionSize Maximum permitted transaction size in bytes.
 * @property modifiedTime ([AutoAcceptable]) Last modification time of network
 ↪parameters set.
 * @property epoch ([AutoAcceptable]) Version number of the network parameters.
 ↪Starting from 1, this will always increment on each new set
 * of parameters.
 * @property whitelistedContractImplementations ([AutoAcceptable]) List of
 ↪whitelisted jars containing contract code for each contract class.
 * This will be used by [net.corda.core.contracts.
↪WhitelistedByZoneAttachmentConstraint].
 * [You can learn more about contract constraints here] (https://docs.corda.net/api-contract-constraints.html).
 * @property packageOwnership ([AutoAcceptable]) List of the network-wide java
 ↪packages that were successfully claimed by their owners.
 * Any Cordapp JAR that offers contracts and states in any of these packages must be
 ↪signed by the owner.
 * @property eventHorizon Time after which nodes will be removed from the network map
 ↪if they have not been seen
 * during this period
 */
@KeepForDJVM
@CordaSerializable
data class NetworkParameters(
    val minimumPlatformVersion: Int,
    val notaries: List<NotaryInfo>,
    val maxMessageSize: Int,
    val maxTransactionSize: Int,
)

```

(continues on next page)

(continued from previous page)

```

    @AutoAcceptable val modifiedTime: Instant,
    @AutoAcceptable val epoch: Int,
    @AutoAcceptable val whitelistedContractImplementations: Map<String, List
    ↵<AttachmentId>>,
    val eventHorizon: Duration,
    @AutoAcceptable val packageOwnership: Map<String, PublicKey>
) {

```

This behaviour can be turned off by setting the optional node configuration property `NetworkParameterAcceptanceSettings.autoAcceptEnabled` to `false`. For example:

```

...
NetworkParameterAcceptanceSettings {
    autoAcceptEnabled = false
}
...

```

It is also possible to switch off this behaviour at a more granular parameter level. This can be achieved by specifying the set of `@AutoAcceptable` parameters that should not be auto-acceptable in the optional `NetworkParameterAcceptanceSettings.excludedAutoAcceptableParameters` node configuration property.

For example, auto-acceptance can be switched off for any updates that change the `packageOwnership` map by adding the following to the node configuration:

```

...
NetworkParameterAcceptanceSettings {
    excludedAutoAcceptableParameters: ["packageOwnership"]
}
...

```

## Manual Acceptance

If the auto-acceptance behaviour is turned off via the configuration or the network parameters change involves parameters that are not auto-acceptable then manual approval is required.

In this case 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 [Node shell](#)):

```
run acceptNewNetworkParameters parametersHash: "ba19fc1b9e9c1c7cbea712efda5f78b53ae4e5d123"
```

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.

### 17.3.5 Cleaning the network map cache

Sometimes it may happen that the node ends up with an inconsistent view of the network. This can occur due to changes in deployment leading to stale data in the database, different data distribution time and mistakes in configuration. For these unlikely events both RPC method and command line option for clearing local network map cache database exist. To use them you either need to run from the command line:

```
java -jar corda.jar clear-network-cache
```

or call RPC method `clearNetworkMapCache` (it can be invoked through the node's shell as `run clearNetworkMapCache`, for more information on how to log into node's shell see [Node shell](#)). As we are testing and hardening the implementation this step shouldn't be required. After cleaning the cache, network map data is restored on the next poll from the server or filesystem.

## 17.4 Cipher suites supported by Corda

### Contents

- *Cipher suites supported by Corda*
  - *Certificate hierarchy*
  - *Supported cipher suites*

The set of signature schemes supported forms a part of the consensus rules for a Corda DLT network. Thus, it is important that implementations do not support pluggability of any crypto algorithms and do take measures to prevent algorithms supported by any underlying cryptography library from becoming accidentally accessible. Signing a transaction with an algorithm that is not a part of the base specification would result in a transaction being considered invalid by peer nodes and thus a loss of consensus occurring. The introduction of new algorithms over time will require a global upgrade of all nodes.

Corda has been designed to be cryptographically agile, in the sense that the available set of signature schemes is carefully selected based on various factors, such as provided security-level and cryptographic strength, compatibility with various HSM vendors, algorithm standardisation, variety of cryptographic primitives, business demand, option for post-quantum resistance, side channel security, efficiency and rigorous testing.

Before we present the pool of supported schemes it is useful to be familiar with [Network certificates](#) and [API: Identity](#). An important design decision in Corda is its shared hierarchy between the TLS and Node Identity certificates.

### 17.4.1 Certificate hierarchy

A Corda network has 8 types of keys and a regular node requires 4 of them:

#### Network Keys

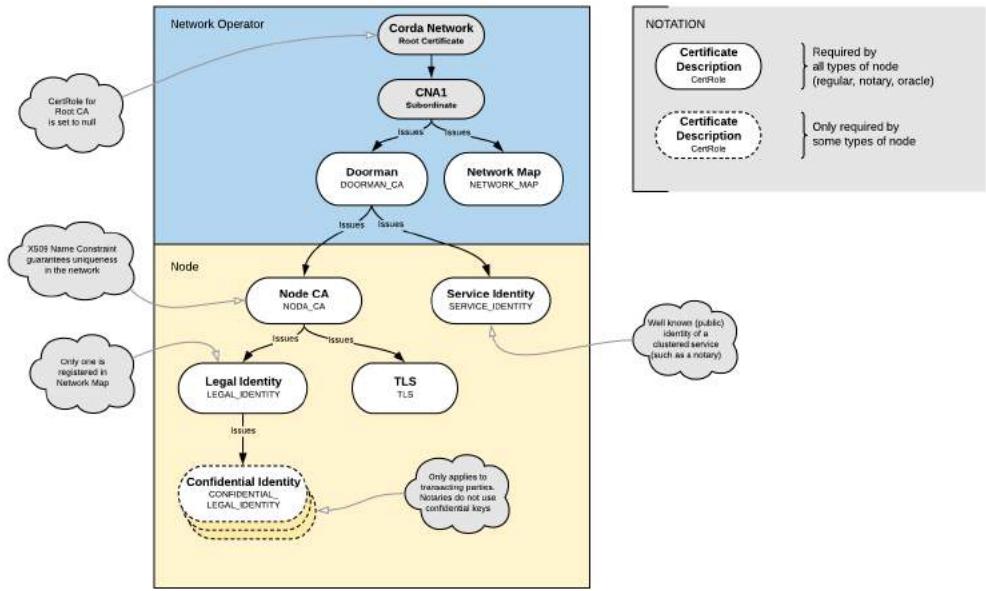
- The **root network CA key**
- The **doorman CA key**
- The **network map key**
- The **service identity key(s)** (per service, such as a notary cluster; it can be a Composite key)

#### Node Keys

- The **node CA key(s)** (one per node)

- The **legal identity** key(s) (one per node)
- The **tls** key(s) (per node)
- The **confidential identity** key(s) (per node)

We can visualise the certificate structure as follows (for a detailed description of cert-hierarchy, see [Network certificates](#)):



#### 17.4.2 Supported cipher suites

Due to the shared certificate hierarchy, the following 4 key/certificate types: **root network CA**, **doorman CA**, **node CA** and **tls** should be compatible with the standard TLS 1.2 protocol. The latter is a requirement from the TLS certificate-path validator. It is highlighted that the rest of the keys can be any of the 5 supported cipher suites. For instance, **network map** is ECDSA NIST P-256 (secp256r1) in the Corda Network (CN) as it is well-supported by the underlying HSM device, but the default for dev-mode is Pure EdDSA (ed25519).

The following table presents the 5 signature schemes currently supported by Corda. The TLS column shows which of them are compatible with TLS 1.2, while the default scheme per key type is also shown in the last column.

Cipher suite	Description	TLS	Default for
Pure EdDSA using the ed25519 curve and SHA-512	<p>EdDSA represents the current state of the art in mainstream cryptography. It implements elliptic curve cryptography with deterministic signatures a fast implementation, explained constants, side channel resistance and many other desirable characteristics. However, it is relatively new and not widely supported, for example, you can't use it in TLS yet (a draft RFC exists but is not standardised yet).</p>	NO	<ul style="list-style-type: none"> <li>node identity</li> <li>confidential identity</li> <li>network map (dev)</li> </ul>
ECDSA using the NIST P-256 curve (secp256r1) and SHA-256	<p>This is the default choice for most systems that support elliptic curve cryptography today and is recommended by NIST. It is also supported by the majority of the HSM vendors.</p>	YES	<ul style="list-style-type: none"> <li>root network CA</li> <li>doorman CA</li> <li>node CA</li> <li>tls</li> <li>network map (CN)</li> </ul>
ECDSA using the Koblitz k1 curve (secp256k1) and SHA-256	<p>secp256k1 is the curve adopted by Bitcoin and as such there is a wealth of infrastructure, code and advanced algorithms designed for use with it. This curve is standardised by NIST as part of the “Suite B” cryptographic algorithms and as such is more widely supported than ed25519.</p>	NO	
504	<p>By supporting it we gain access to the ecosystem of advanced cryptographic techniques</p>		<b>Chapter 17. Networks</b>

## 17.5 Joining an existing compatibility zone

To connect to a compatibility zone you need to register with its certificate signing authority (or *doorman*) by submitting a certificate signing request (CSR) to obtain a valid identity for the zone. This process is only necessary when the node connects to the network for the first time, or when the certificate expires. You could do this out of band, for instance via email or a web form, but there's also a simple request/response utility built into the node.

Before using this utility, you must first have received the trust store file containing the root certificate from the zone operator. For high security zones, this might be delivered physically. Then run the following command:

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

By default, the utility expects the trust store file to be in the location `certificates/network-root-truststore.jks`. This can be overridden using the additional `--network-root-truststore` flag.

The utility performs the following steps:

1. It creates a certificate signing request based on the following information from the node's configuration file (see [Node configuration](#)):
  - **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. You can use the X.500 schema to disambiguate entities that have the same or similar brand names
  - **emailAddress** e.g. “[admin@company.com](mailto:admin@company.com)”
  - **devMode** must be set to false
  - **compatibilityZoneURL** or **networkServices** The address(es) used to register with the compatibility zone and retrieve the network map. These should be provided to you by the operator of the zone. This must be either:
    - **compatibilityZoneURL** The root address of the network management service. Use this if both the doorman and the network map service are operating on the same URL endpoint
    - **networkServices** The root addresses of the doorman and the network map service. Use this if the doorman and the network map service are operating on the same URL endpoint, where:
      - \* **doormanURL** is the root address of the doorman. This is the address used for initial registration
      - \* **networkMapURL** is the root address of the network map service
2. It generates a new private/public keypair to sign the certificate signing request
3. It submits the request to the doorman server and polls periodically to retrieve the corresponding certificates
4. It creates the node's keystore and trust store using the received certificates
5. It creates and stores the node's TLS keys and legal identity key along with their corresponding certificate-chains

---

**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 as long as the `--initial-registration` flag is specified.

---

## 17.6 Joining Corda Testnet

### Contents

- *Joining Corda Testnet*
  - *Deploying a Corda node to the Corda Testnet*
  - *A note on identities on Corda Testnet*

The Corda Testnet is an open public network of Corda nodes on the internet. It is designed to be a complement to the Corda Network where any entity can transact real world value with any other counterparty in the context of any application. The Corda Testnet is designed for “non-production” use in a genuine global context of Corda nodes, including but not limited to CorDapp development, multi-party testing, demonstration and showcasing of applications and services, learning, training and development of the Corda platform technology and specific applications of Corda.

The Corda Testnet is based on exactly the same technology as the main Corda Network, but can be joined on a self-service basis through the automated provisioning system described below.

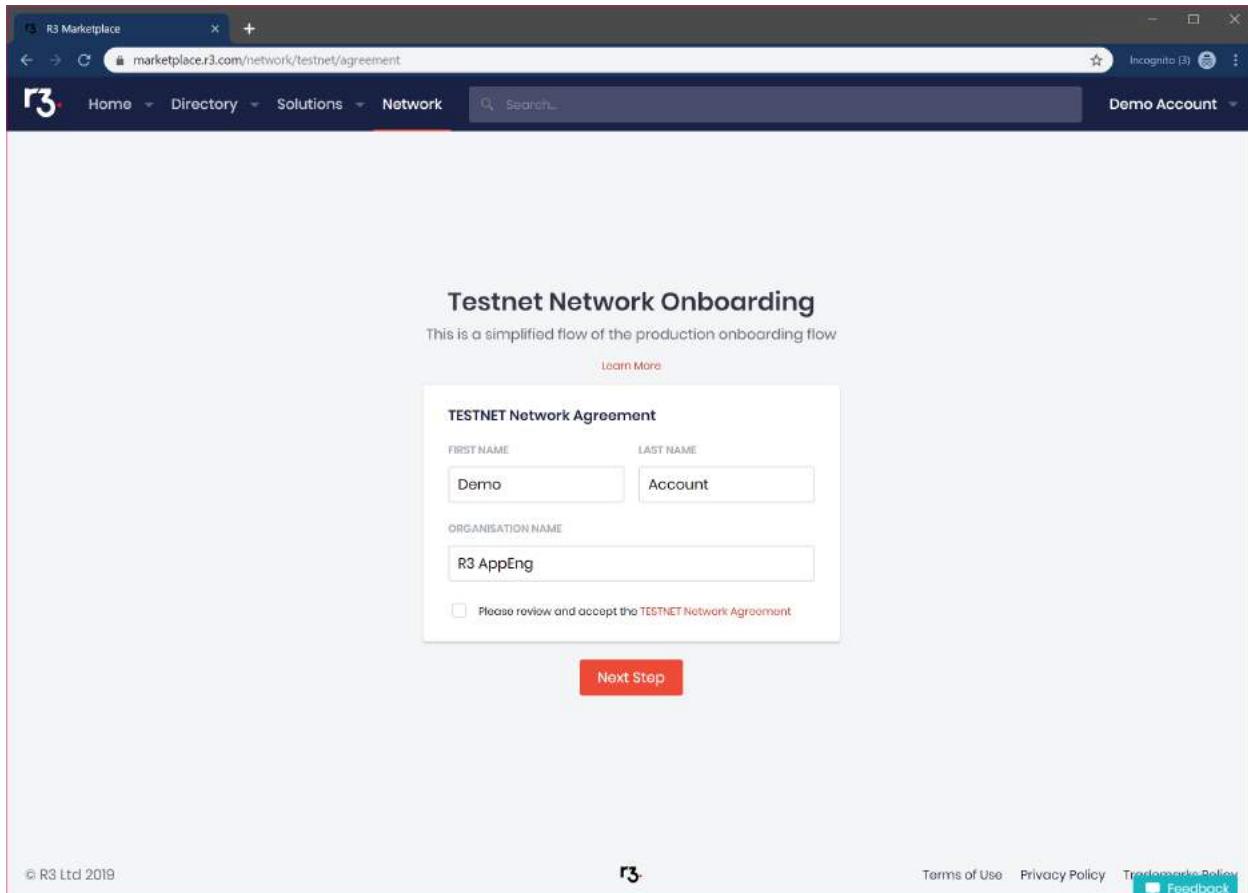
### 17.6.1 Deploying a Corda node to the Corda Testnet

The screenshot shows the R3 Marketplace dashboard. At the top, there's a navigation bar with links for Home, Directory, Solutions, and Network. Below this, a large red banner titled "Networks on Corda" encourages users to discover a worldwide community of users, applications, and services. A link to "Learn more about networks on Corda" is provided. In the bottom right corner, there's a section titled "Testing Networks" which lists two options: "Testnet Network" and "Cordite Network". Each option has a brief description, a governance link, and a "Join" button.

The Corda Testnet is accessible via <https://marketplace.r3.com/>. Click on “Join the Corda Testnet” to begin joining the network.

This will create an account with the Testnet on-boarding application which will enable you to provision and manage multiple Corda nodes on Testnet. You will log in to this account to view and manage your Corda Testnet identity

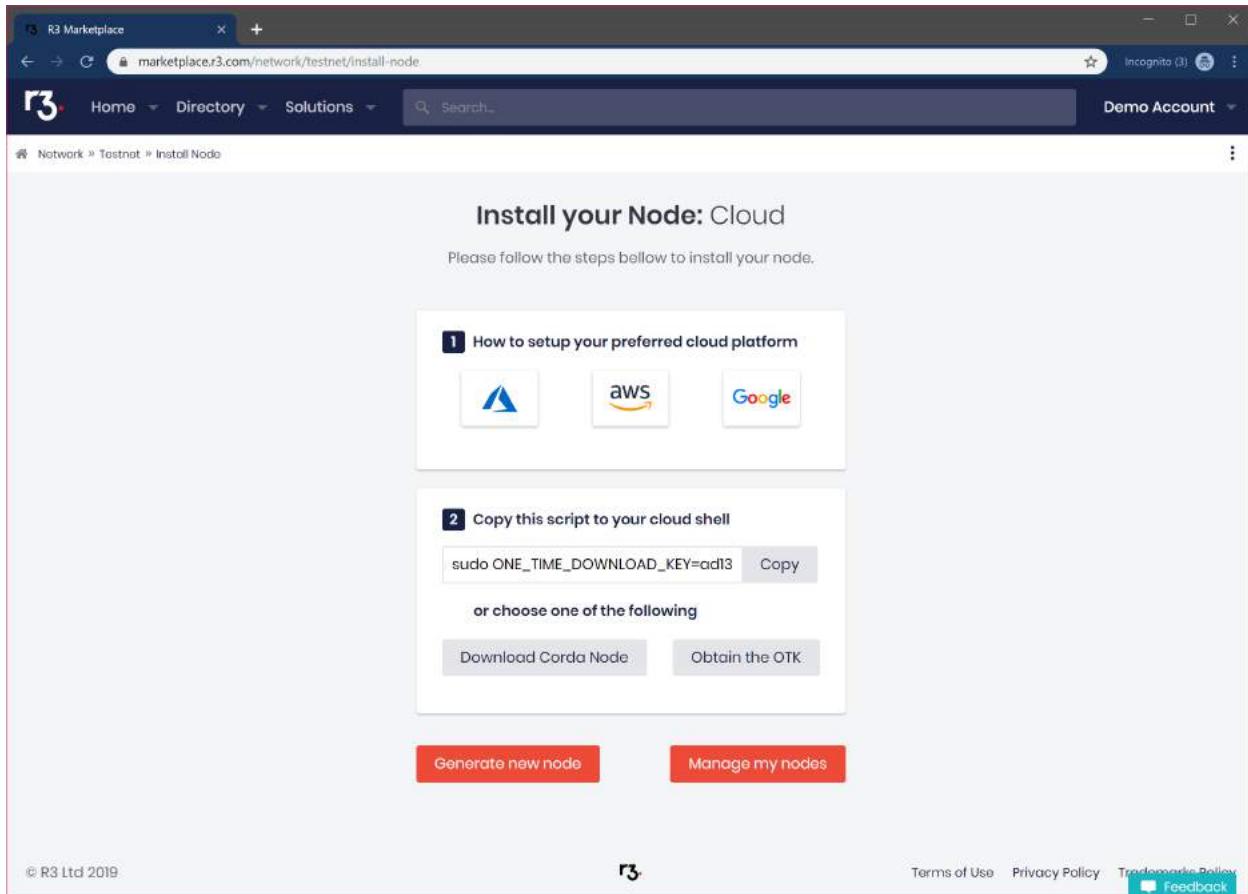
certificates.



Fill in the registration form and please read and accept the terms of use, then click Next Step; this will take you to the Testnet dashboard where you can see a list of your nodes as well as create new ones.

When creating a new node you can now choose how to deploy it to the Corda Testnet. We strongly recommend hosting your Corda node on a public cloud resource.

Select the cloud provider you wish to use for documentation on how to specifically configure Corda for that environment.



Once your cloud instance is set up you can install and run your Testnet pre-provisioned Corda node by clicking on “Copy” and pasting the one time link into your remote cloud terminal.

The installation script will download the Corda binaries as well as your PKI certificates, private keys and supporting files and will install and run Corda on your fresh cloud VM. Your node will register itself with the Corda Testnet when it first runs and be added to the global network map and be visible to counterparties after approximately 5 minutes.

Hosting a Corda node locally is possible but will require manually configuring firewall and port forwarding on your local router. If you want this option then click on the “Download” button to download a Zip file with a pre-configured Corda node.

**Note:** If you host your node on your own machine or a corporate server you must ensure it is reachable from the public internet at a specific IP address. Please follow the instructions here: *Deploying Corda to Corda Testnet from your local environment*.

### 17.6.2 A note on identities on Corda Testnet

Unlike the main Corda Network, which is designed for verified real world identities, The Corda Testnet automatically assigns a “distinguished name” as your identity on the network. This is to prevent name abuse such as the use of offensive language in the names or name squatting. This allows the provision of a node to be automatic and instantaneous. It also enables the same user to safely generate many nodes without accidental name conflicts. If you require a human readable name then please contact support and a partial organisation name can be approved.

## 17.7 Deploying Corda to Testnet

### 17.7.1 Deploying Corda to Corda Testnet from an Azure Cloud Platform VM

#### Contents

- *Deploying Corda to Corda Testnet from an Azure Cloud Platform VM*
  - *Pre-requisites*
  - *Deploy Corda node*
    - \* *STEP 1: Create a Resource Group*
    - \* *STEP 2: Launch the VM*
    - \* *STEP 3: Connect to your VM and set up the environment*
    - \* *STEP 4: Download and set up your Corda node*
  - *Testing your deployment*

This document will describe how to set up a virtual machine on the Azure Cloud Platform to deploy your pre-configured Corda node and automatically connect to Testnet. A self-service download link can be obtained from <https://marketplace.r3.com/network/testnet>.

#### Pre-requisites

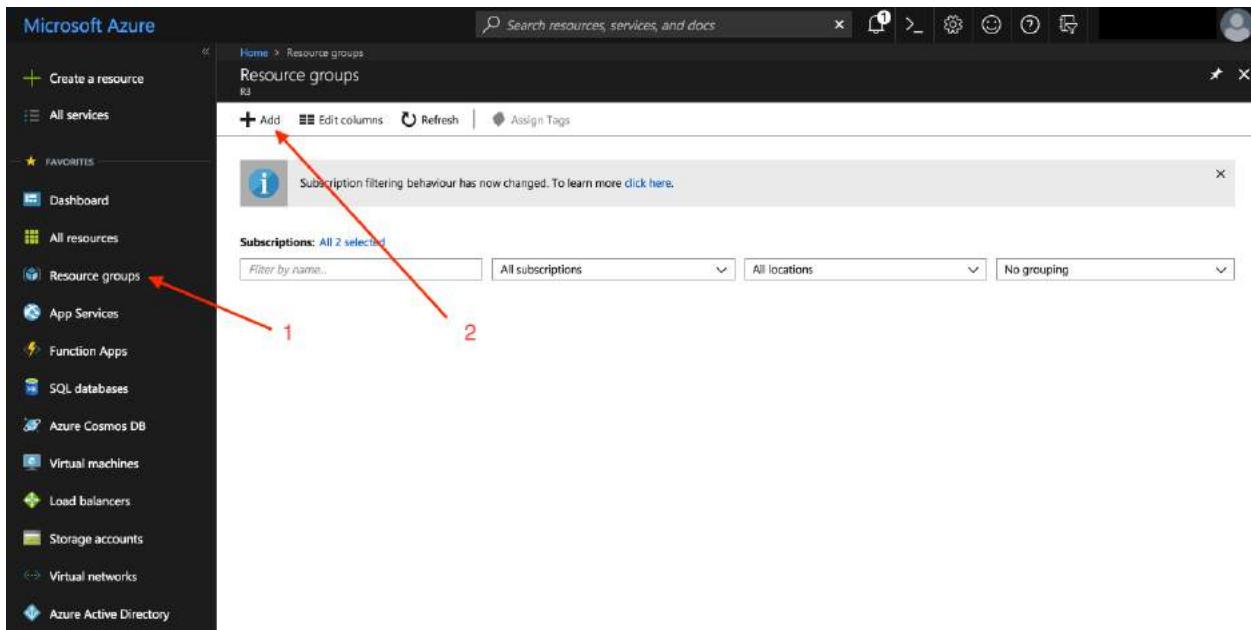
- Ensure you have a registered Microsoft Azure account which can create virtual machines.

#### Deploy Corda node

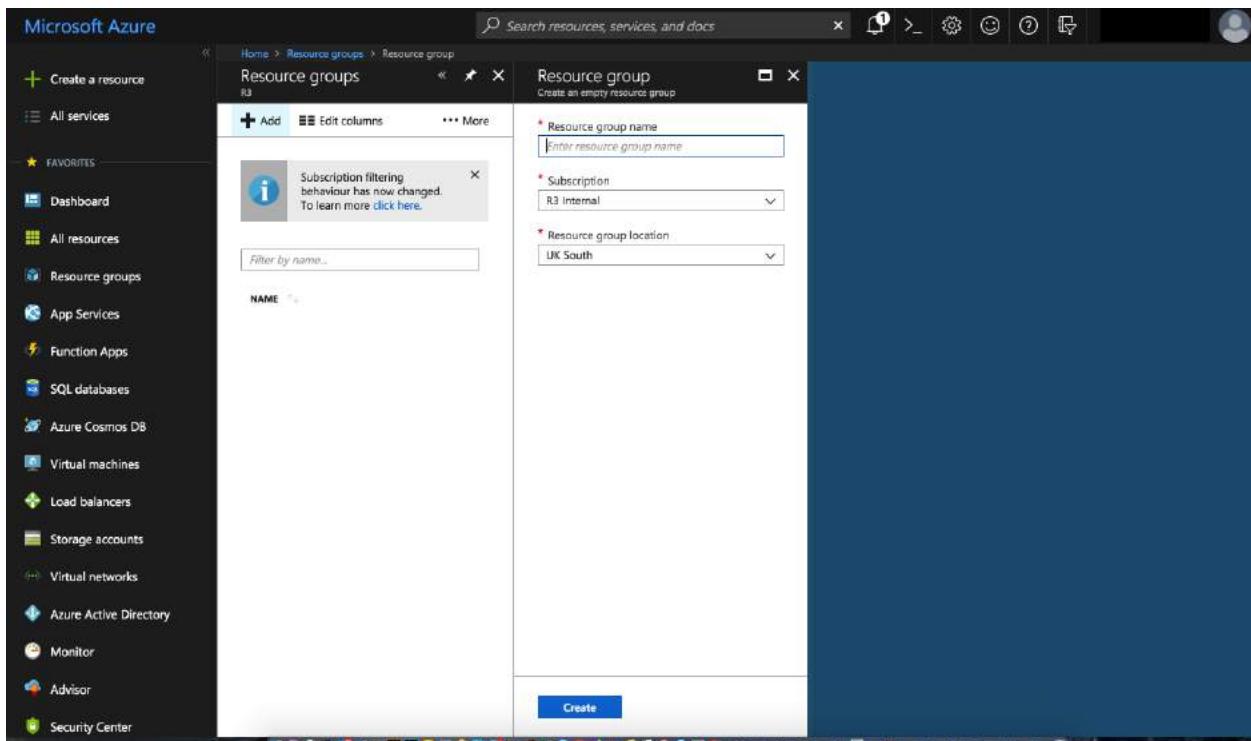
Browse to <https://portal.azure.com> and log in with your Microsoft account.

#### STEP 1: Create a Resource Group

Click on the “Resource groups” link in the side nav in the Azure Portal and then click “Add”:



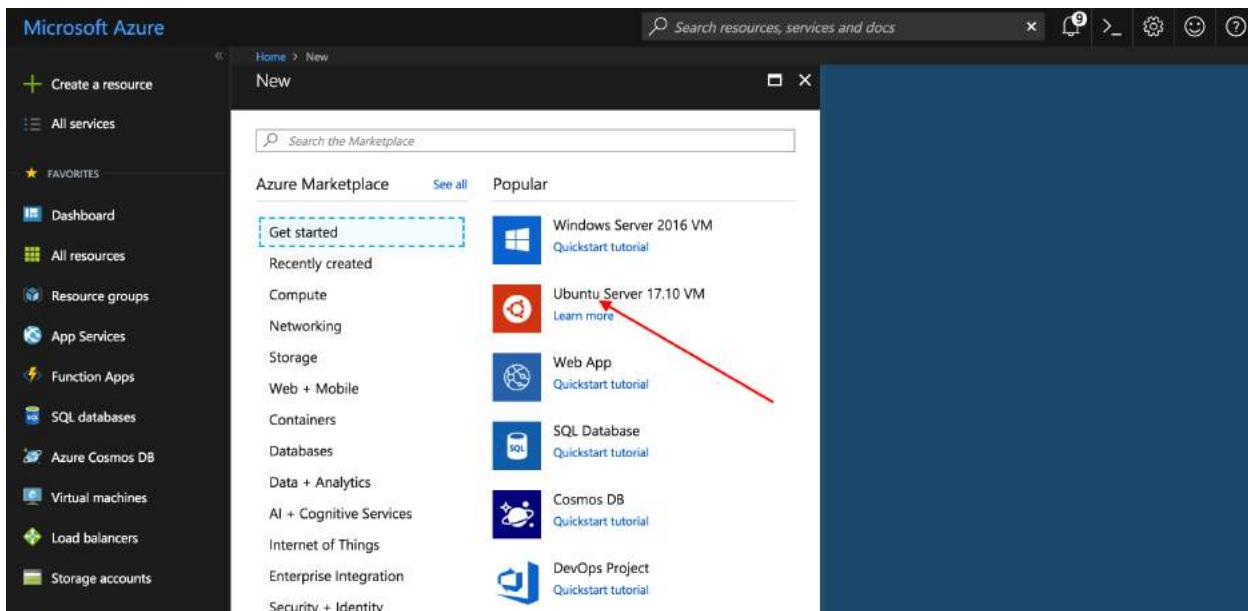
Fill in the form and click “Create”:



### STEP 2: Launch the VM

At the top of the left sidenav click on the button with the green cross “Create a resource”.

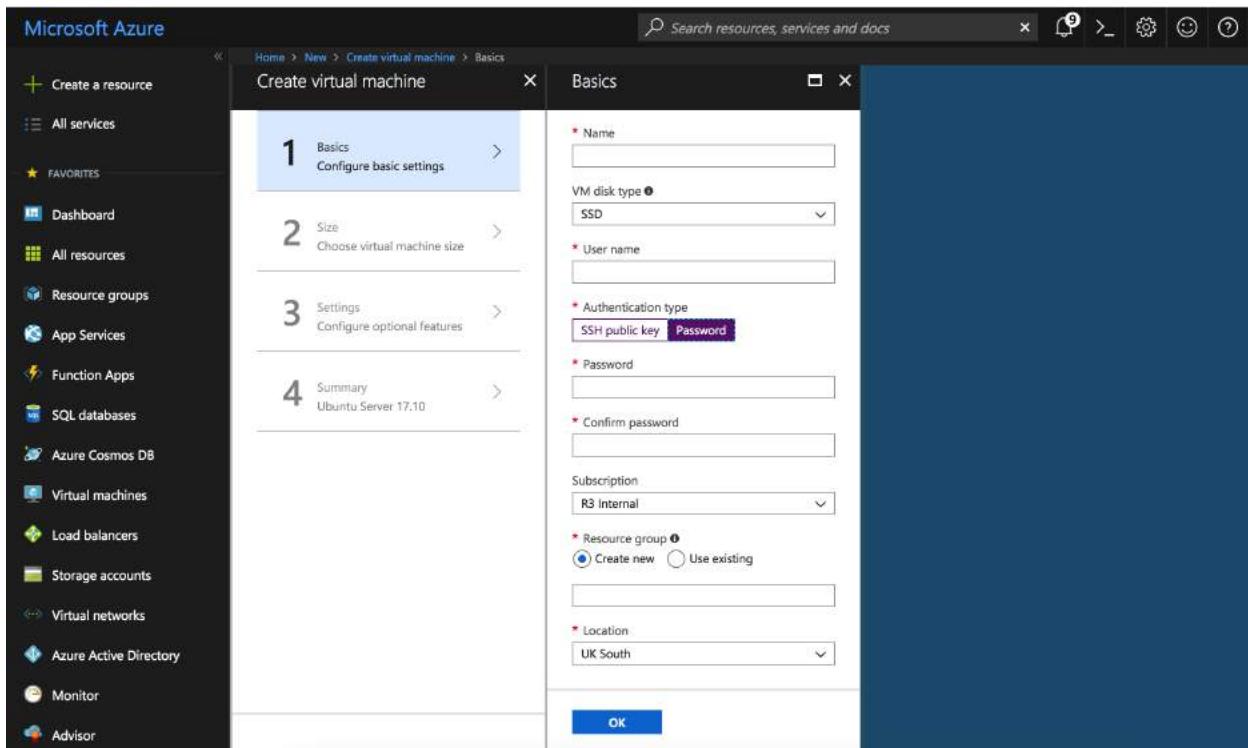
In this example we are going to use an Ubuntu server so select the latest Ubuntu Server option:



Fill in the form:

- Add a username (to log into the VM) and choose and enter a password
- Choose the resource group we created earlier from the “Use existing” dropdown
- Select a cloud region geographically near to your location to host your VM

Click on “OK”:



Choose a size (“D4S\_V3 Standard” is recommended if available) and click “Select”:

The screenshot shows the Microsoft Azure interface for creating a new virtual machine. The left sidebar includes options like 'Create a resource', 'All services', and 'Favorites'. The main area is titled 'Choose a size' and lists various VM sizes under 'Available'. The selected size is D4s\_v3, which has 4 vCPUs, 8 GB RAM, 8 data disks, and 4000 MAX IOPS. The price is listed as US\$172.61. A note at the bottom states: 'Prices presented are estimates in your local currency that include only Azure infrastructure costs and any discounts for the subscription and location. The prices don't include any applicable software costs. Recommended sizes are determined by the publisher of the selected image based on hardware and software requirements.' A blue 'Select' button is at the bottom.

Click on “Public IP address” to open the “Settings” panel

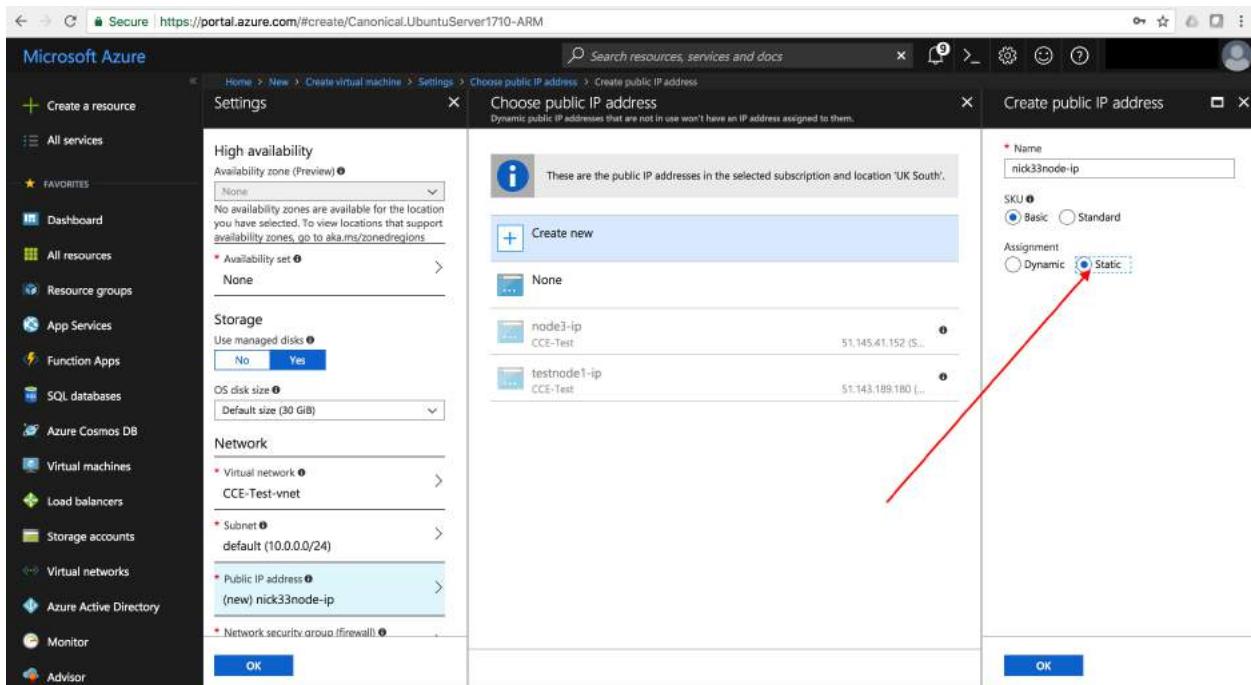
The screenshot shows the 'Create virtual machine' settings panel. Step 3, 'Settings', is currently selected. On the right, under the 'Network' section, there is a list of network configurations. One item, 'Public IP address (new) nick33node-ip', is highlighted with a red arrow pointing to it. The 'OK' button is visible at the bottom of the settings panel.

Set the IP address to “Static” under “Assignment” and click “OK”:

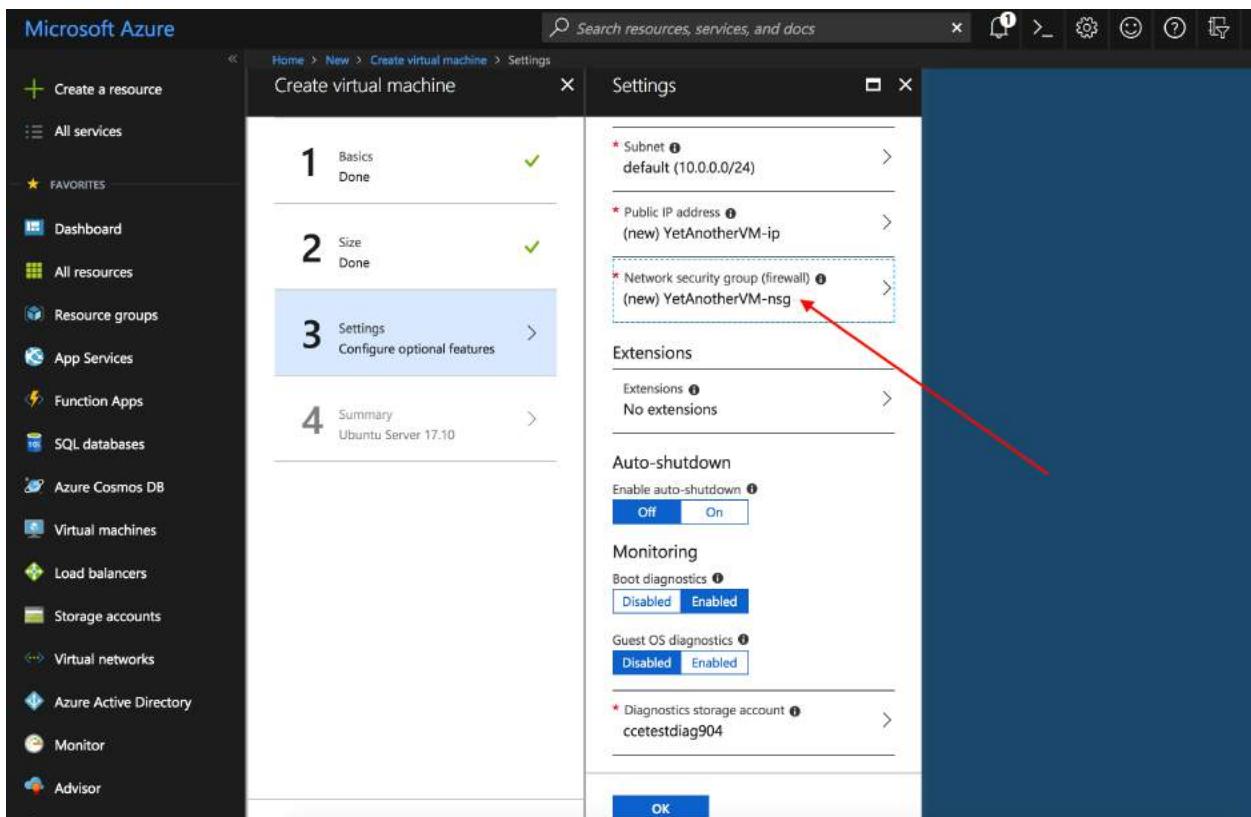
---

**Note:** This is so the IP address for your node does not change frequently in the global network map.

---



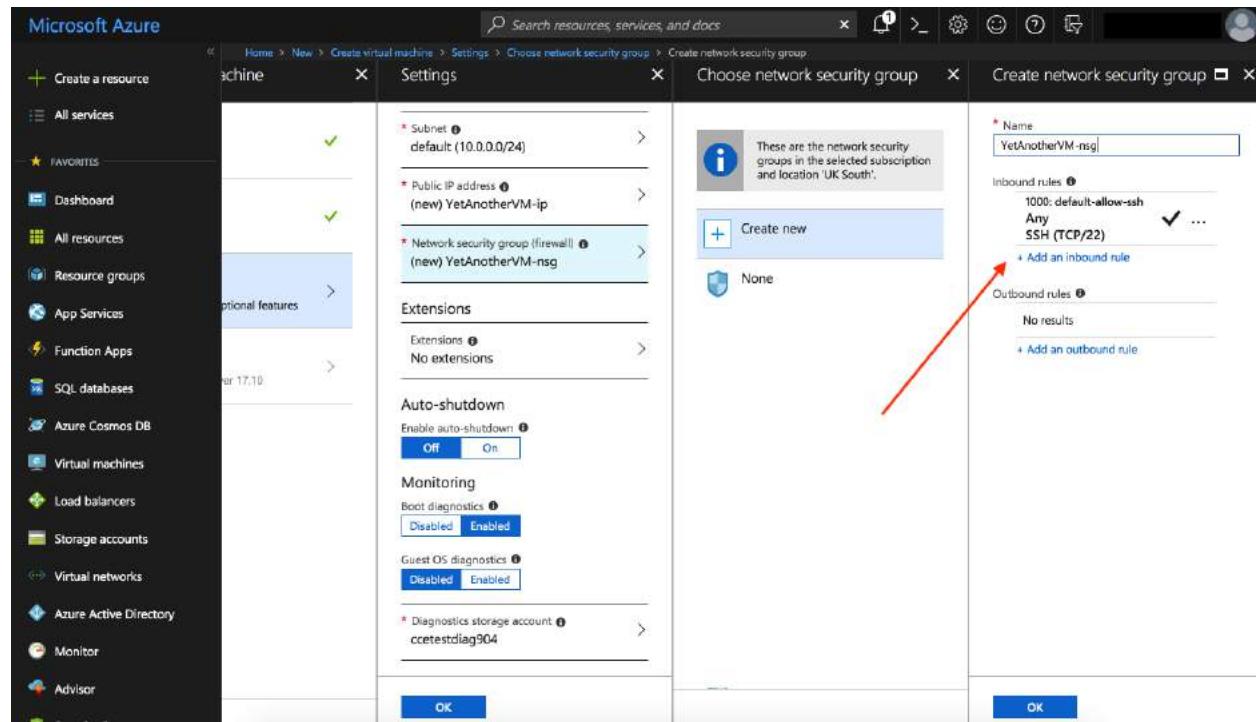
Next toggle “Network Security Group” to advanced and click on “Network security group (firewall)”:



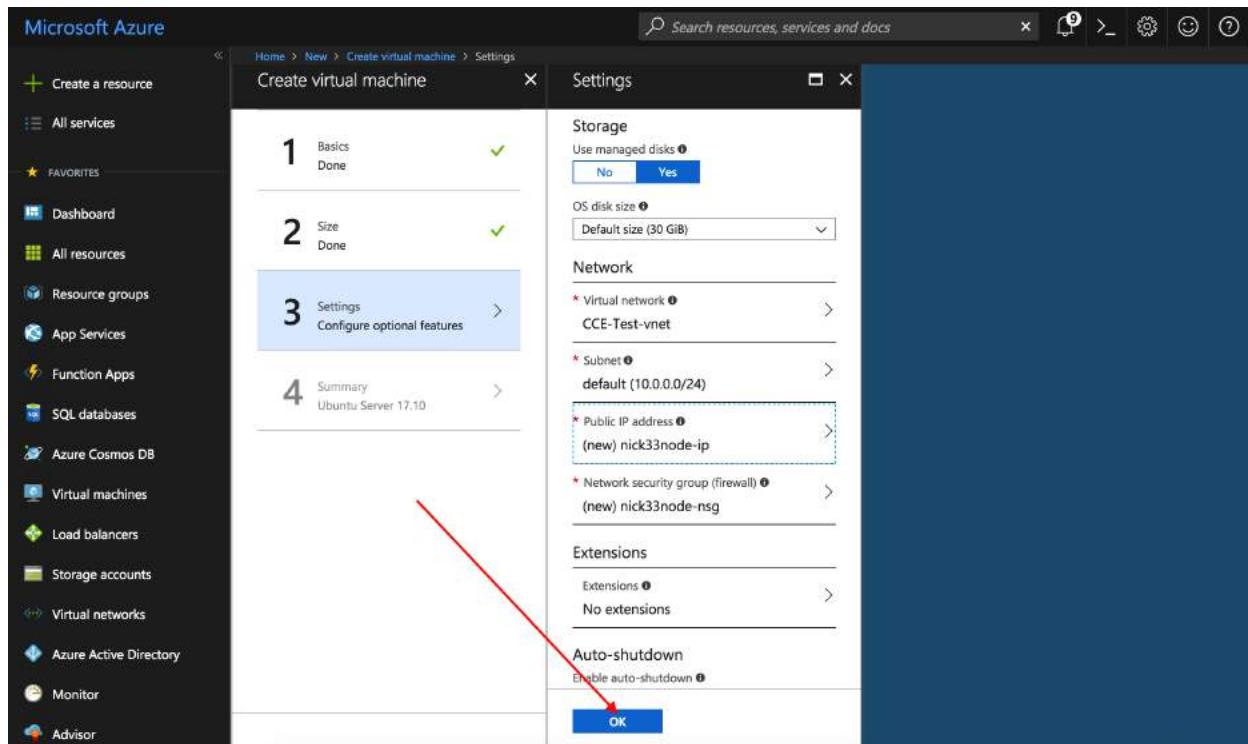
Add the following inbound rules for ports 8080 (webserver), and 10002-10003 for the P2P and RPC ports used by the Corda node respectively:

```
Destination port ranges: 10002, Priority: 1041 Name: Port_10002
Destination port ranges: 10003, Priority: 1042 Name: Port_10003
Destination port ranges: 8080, Priority: 1043 Name: Port_8080
Destination port ranges: 22, Priority: 1044 Name: Port_22
```

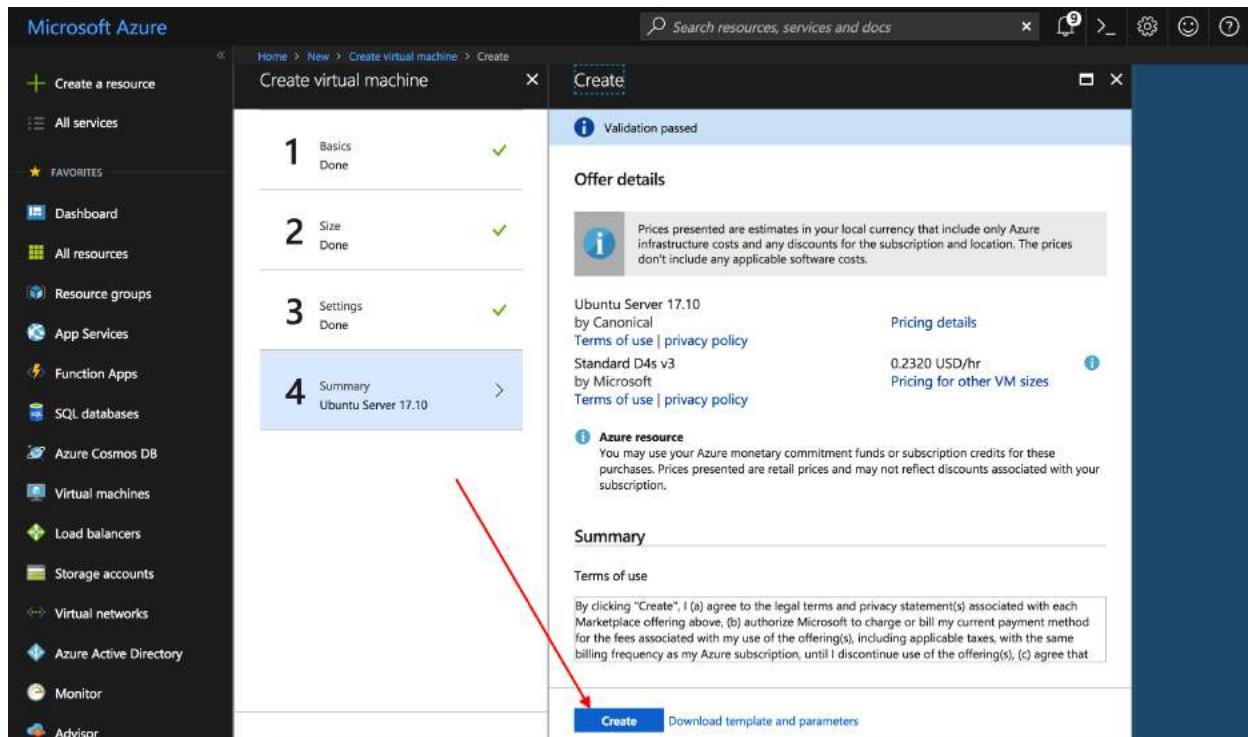
**Note:** The priority has to be unique number in the range 900 (highest) and 4096 (lowest) priority. Make sure each rule has a unique priority or there will be a validation failure and error message.



Click “OK” and “OK” again on the “Settings” panel:

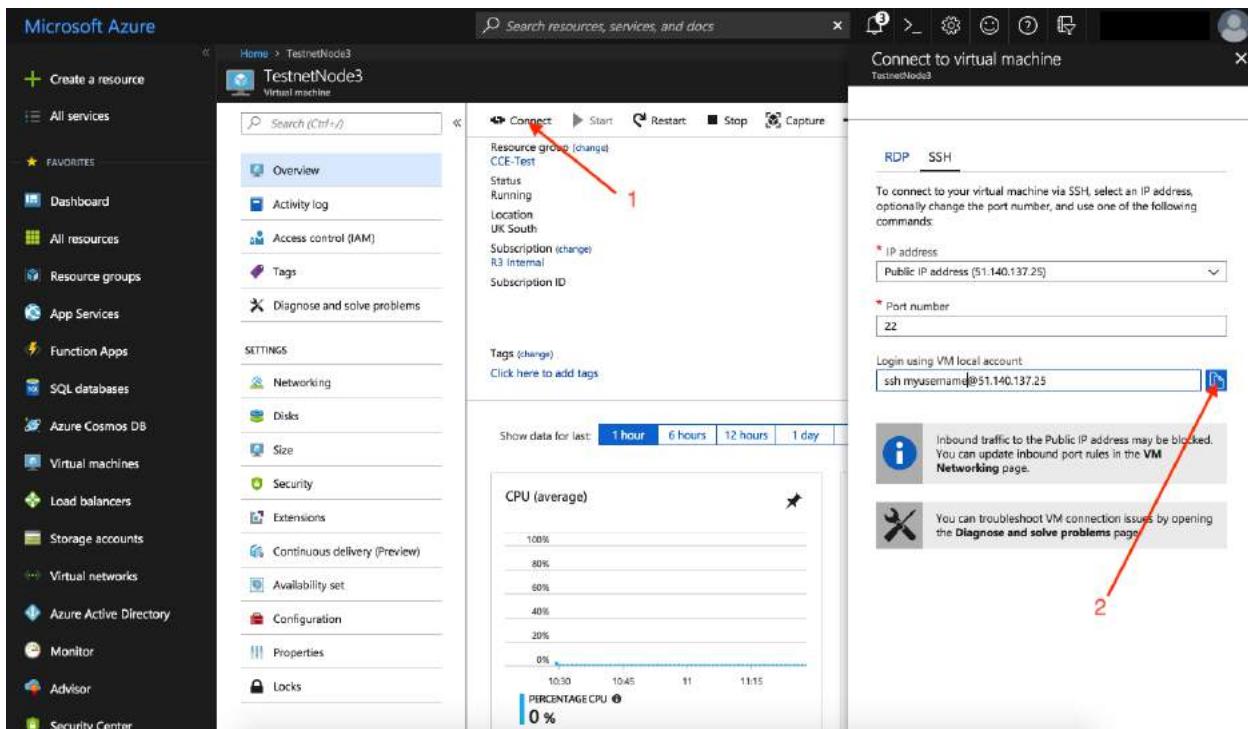


Click “Create” and wait a few minutes for your instance to be provisioned and start running:



### STEP 3: Connect to your VM and set up the environment

Once your instance is running click on the “Connect” button and copy the ssh command:



Enter the ssh command into your terminal. At the prompt, type “yes” to continue connecting and then enter the password you configured earlier to log into the remote VM:

```
Last login: Mon Mar 19 11:22:38 on ttys001
[admins-MacBook-Pro:~ nickarini$ ssh nickarini@51.145.11.116
The authenticity of host '51.145.11.116 (51.145.11.116)' can't be established.
ECDSA key fingerprint is SHA256:FXxXz38hC83RVXYu0BfUyc2e4tes3TPNda+MUDkE2pQ.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '51.145.11.116' (ECDSA) to the list of known hosts.
[nickarini@51.145.11.116's password:
Permission denied, please try again.
[nickarini@51.145.11.116's password:
Welcome to Ubuntu 17.10 (GNU/Linux 4.13.0-37-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 Get cloud support with Ubuntu Advantage Cloud Guest:
  nos http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

working
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <commands>".
See "man sudo_root" for details.

nickarini@nick33node:~$
```

## STEP 4: Download and set up your Corda node

Now that your Azure environment is configured you can switch to the [Testnet dashboard](#) and click “Copy” to get a one-time installation script.

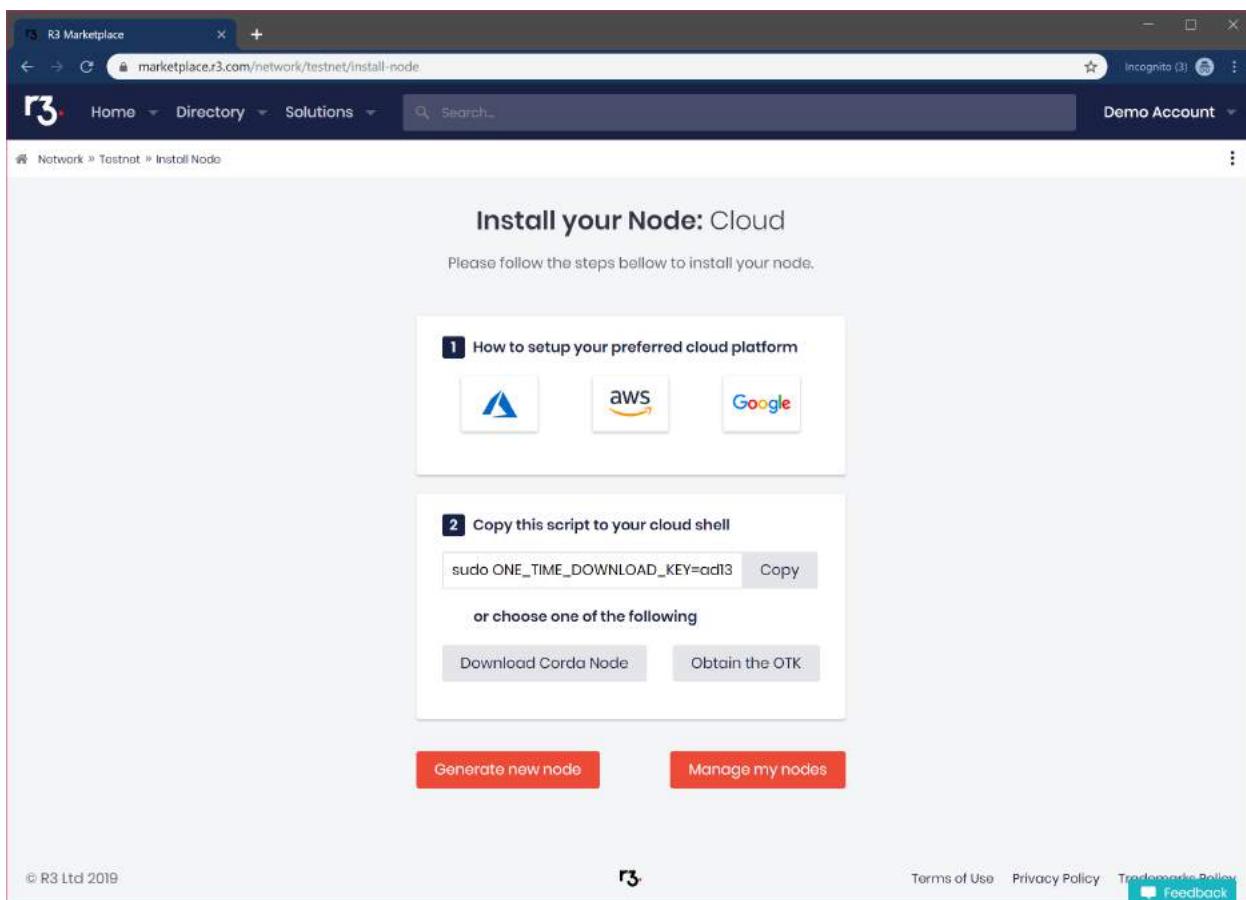
---

**Note:** If you have not already set up your account on Testnet, please visit <https://marketplace.r3.com/network/testnet> and sign up.

---

**Note:** You can generate as many Testnet identities as you like by clicking “Generate new node” to generate a new one-time link.

---



The screenshot shows the R3 Marketplace Testnet Install Node page. At the top, there's a header with the R3 logo, navigation links (Home, Directory, Solutions), a search bar, and a Demo Account dropdown. Below the header, the URL is marketplace.r3.com/network/testnet/install-node. The main content area has a title "Install your Node: Cloud" and a sub-instruction "Please follow the steps below to install your node." It lists two steps: 1. How to setup your preferred cloud platform, with icons for Azure, AWS, and Google. 2. Copy this script to your cloud shell, displaying the command "sudo ONE\_TIME\_DOWNLOAD\_KEY=cd13" and a "Copy" button, followed by "or choose one of the following" with "Download Corda Node" and "Obtain the OTK" buttons. At the bottom are "Generate new node" and "Manage my nodes" buttons, along with footer links for Terms of Use, Privacy Policy, Trademarks Policy, and Feedback.

In the terminal of your cloud instance, paste the command you just copied to install and run your Corda node:

```
sudo ONE_TIME_DOWNLOAD_KEY=YOUR_UNIQUE_DOWNLOAD_KEY_HERE bash -c "$(curl -L https://onboarder.prod.ws.r3.com/api/user/node/TESTNET/install.sh)"
```

**Warning:** This command will execute the install script as ROOT on your cloud instance. You may wish to examine the script prior to executing it on your machine.

You can follow the progress of the installation by typing the following command in your terminal:

```
tail -f /opt/corda/logs/node-<VM-NAME>.log
```

## Testing your deployment

To test that your deployment is working correctly, follow the instructions in [Using the Node Explorer to test a Corda node on Corda Testnet](#) to set up the Finance CorDapp and issue cash to a counterparty.

This will also demonstrate how to install a custom CorDapp.

### 17.7.2 Deploying Corda to Corda Testnet from an AWS Cloud Platform VM

#### Contents

- *Deploying Corda to Corda Testnet from an AWS Cloud Platform VM*
  - *Pre-requisites*
  - *Deploy Corda node*
  - *Testing your deployment*

This document explains how to deploy a Corda node to AWS that can connect directly to the Corda Testnet. A self service download link can be obtained from <https://marketplace.r3.com/network/testnet>. This document will describe how to set up a virtual machine on the AWS Cloud Platform to deploy your pre-configured Corda node and automatically connect to Testnet.

#### Pre-requisites

- Ensure you have a registered Amazon AWS account which can create virtual machines and you are logged on to the AWS console: <https://console.aws.amazon.com>.

#### Deploy Corda node

Browse to <https://console.aws.amazon.com> and log in with your AWS account.

##### STEP 1: Launch a new virtual machine

Click on Launch a virtual machine with EC2.

## Build a solution

Get started with simple wizards and automated workflows.



In the quick start wizard scroll down and select the most recent Ubuntu machine image as the Amazon Machine Image (AMI).

Step 1: Choose an Amazon Machine Image (AMI)

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   7. Review   Cancel and Exit   Select

Free tier only

<b>Amazon Linux</b> Free tier eligible	Amazon Linux 2 LTS Candidate 2 AMI (HVM), SSD Volume Type - ami-f973ab84	64-bit
<b>Red Hat</b> Free tier eligible	Red Hat Enterprise Linux 7.4 (HVM), SSD Volume Type - ami-26ebbc5c	64-bit
<b>SUSE Linux</b> Free tier eligible	SUSE Linux Enterprise Server 12 SP3 (HVM), SSD Volume Type - ami-62bda218	64-bit
<b>Ubuntu Server 16.04 LTS</b> (HVM), SSD Volume Type - ami-43a15f3e Free tier eligible	Ubuntu Server 16.04 LTS (HVM), EBS General Purpose (SSD) Volume Type. Support available from Canonical ( <a href="http://www.ubuntu.com/cloud/services">http://www.ubuntu.com/cloud/services</a> ). Root device type: ebs   Virtualization type: hvm   ENA Enabled: Yes	64-bit

Select the instance type (for example t2.xlarge).

Step 2: Choose an Instance Type

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   7. Review

Currently selected: t2.xlarge (Variable ECUs, 4 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 16 GiB memory, EBS only)

Filter by:	All instance types	Current generation	Show/Hide Columns					
	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
<input type="checkbox"/>	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.micro	1	1	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes
<input checked="" type="checkbox"/>	General purpose	<b>t2.xlarge</b>	4	16	EBS only	-	Moderate	Yes
<input type="checkbox"/>	General purpose	t2.2xlarge	8	32	EBS only	-	Moderate	Yes
<input type="checkbox"/>	General purpose	m5.large	2	8	EBS only	Yes	Up to 10 Gigabit	Yes

Cancel   Previous   **Review and Launch**   Next: Configure Instance Details

Configure a couple of other settings before we review and launch

Under the storage tab (Step 4) increase the storage to 40GB:

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   7. Review

**Step 4: Add Storage**  
Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. [Learn more](#) about storage options in Amazon EC2.

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encrypted
Root	/dev/sda1	snap-0a83a22928e9afbe9	40	General Purpose SSD (GP2)	120 / 3000	N/A	<input checked="" type="checkbox"/>	Not Encrypted

[Add New Volume](#)

Configure the security group (Step 6) to open the firewall ports which Corda uses.

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   7. Review

**Step 6: Configure Security Group**  
A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group:  Create a new security group  
 Select an existing security group

Security group name: launch-wizard-2  
Description: launch-wizard-2 created 2018-04-10T16:24:52.994+01:00

Type	Protocol	Port Range	Source	Description
SSH	TCP	22	Custom 0.0.0.0/0	e.g. SSH for Admin Desktop
Custom TCP	TCP	10002-10003	Anywhere 0.0.0.0/0, ::/0	Corda Node p2p and RPC
Custom TCP	TCP	8080	Anywhere 0.0.0.0/0, ::/0	Webserver

Add a firewall rule for port range 10002-10003 and allow connection from Anywhere. Add another rule for the webserver on port 8080.

Click on the Review and Launch button then if everything looks ok click Launch.

You will be prompted to set up keys to securely access the VM remotely over ssh. Select “Create a new key pair” from the drop down and enter a name for the key file. Click download to get the keys and keep them safe on your local machine.

---

**Note:** These keys are just for connecting to your VM and are separate from the keys Corda will use to sign transactions. These keys will be generated as part of the download bundle.

---

**Select an existing key pair or create a new key pair**

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

Create a new key pair

Key pair name

MyNewKeyPair

Download Key Pair

You have to download the **private key file** (\*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

Cancel Launch Instances

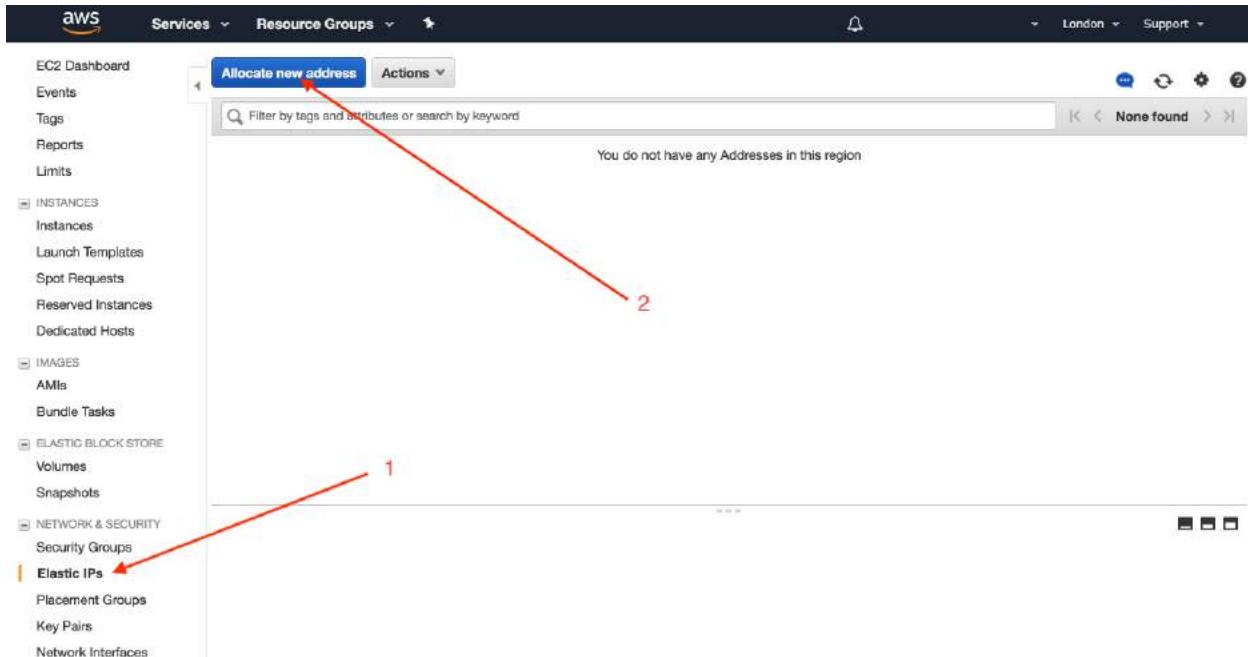
Click “Launch Instances”.

Click on the link to go to the Instances pages in the AWS console where after a few minutes you will be able to see your instance running.

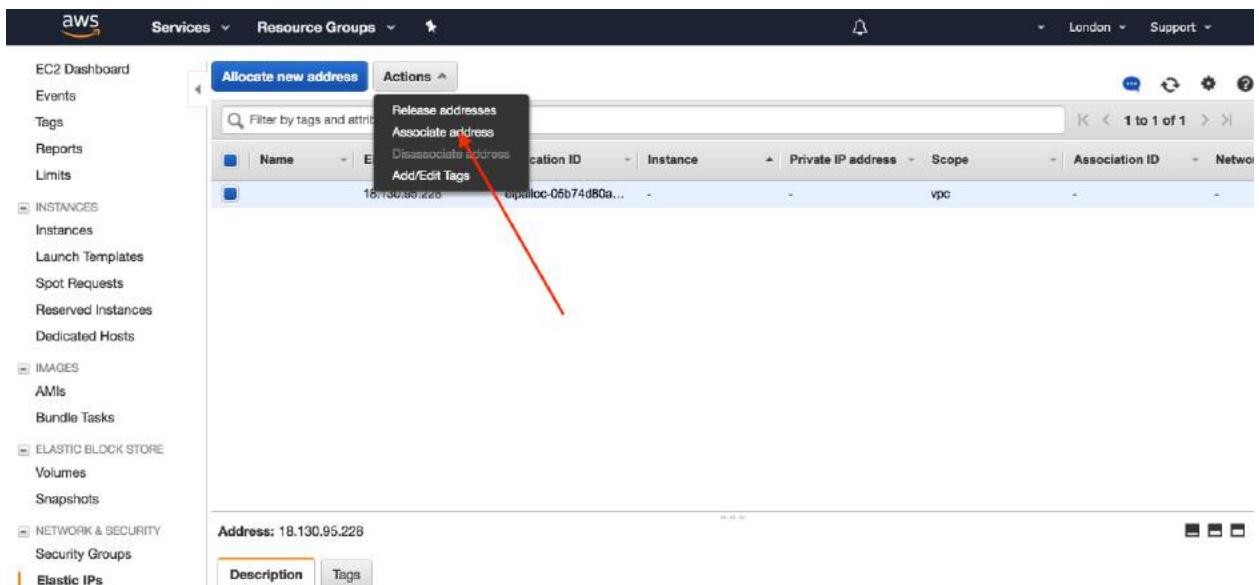
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)
	i-06e3ed205b3ffe25b	t2.xlarge	us-east-1c	running	2/2 checks ...	None	ec2-34-229-21-149.compute-1.amazonaws.c...

## STEP 2: Set up static IP address

On AWS a permanent IP address is called an Elastic IP. Click on the “Elastic IP” link in the navigation panel on the left hand side of the console and then click on “Allocate new address”:



Follow the form then once the address is allocated click on “Actions” then “Associate address”:



Then select the instance you created for your Corda node to attach the IP address to.

### STEP 3: Connect to your VM and set up the environment

In the instances console click on “Connect” and follow the instructions to connect to your instance using ssh.



## Connect To Your Instance



I would like to connect with

A standalone SSH client

A Java SSH Client directly from my browser (Java required)

### To access your instance:

1. Open an SSH client. (find out how to [connect using PuTTY](#))
2. Locate your private key file (myaws.pem). The wizard automatically detects the key you used to launch the instance.
3. Your key must not be publicly viewable for SSH to work. Use this command if needed:

```
chmod 400 myaws.pem
```

4. Connect to your instance using its Public DNS:

```
ec2-34-229-21-149.compute-1.amazonaws.com
```

### Example:

```
ssh -i "myaws.pem" ubuntu@ec2-34-229-21-149.compute-1.amazonaws.com
```

Please note that in most cases the username above will be correct, however please ensure that you read your AMI usage instructions to ensure that the AMI owner has not changed the default AMI username.

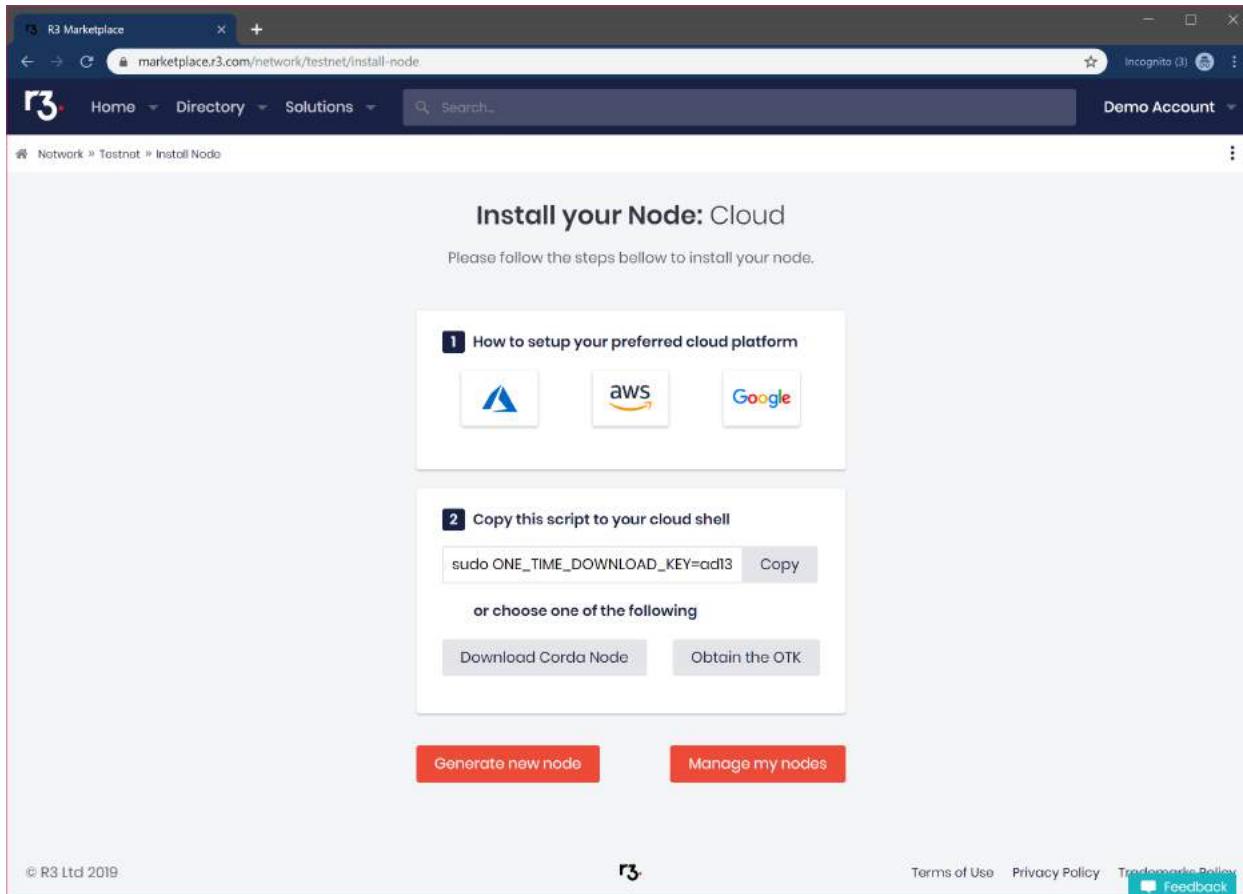
If you need any assistance connecting to your instance, please see our [connection documentation](#).

**Close**

## STEP 4: Download and set up your Corda node

Now your AWS environment is configured you can switch back to the Testnet web application and click on the copy to clipboard button to get a one time installation script.

**Note:** If you have not already set up your account on Testnet then please visit <https://marketplace.r3.com/network/testnet> and sign up.



You can generate as many Testnet identities as you like by refreshing this page to generate a new one time link.

In the terminal of your cloud instance paste the command you just copied to install and run your unique Corda instance on that instance:

```
sudo ONE_TIME_DOWNLOAD_KEY=YOUR_UNIQUE_DOWNLOAD_KEY_HERE bash -c "$(curl -L https://onboarder.prod.ws.r3.com/api/user/node/TESTNET/install.sh)"
```

**Warning:** This command will execute the install script as ROOT on your cloud instance. You may wish to examine the script prior to executing it on your machine.

You can follow the progress of the installation by typing the following command in your terminal:

```
tail -f /opt/corda/logs/node-<VM-NAME>.log
```

### Testing your deployment

To test your deployment is working correctly follow the instructions in [Using the Node Explorer to test a Corda node on Corda Testnet](#) to set up the Finance CorDapp and issue cash to a counterparty.

This will also demonstrate how to install a custom CorDapp.

### 17.7.3 Deploying Corda to Corda Testnet from a Google Cloud Platform VM

#### Contents

- *Deploying Corda to Corda Testnet from a Google Cloud Platform VM*
  - *Pre-requisites*
  - *Deploy Corda node*
  - *Testing your deployment*

This document explains how to deploy a Corda node to Google Cloud Platform that can connect directly to the Corda Testnet. A self service download link can be obtained from <https://marketplace.r3.com/network/testnet>. This document will describe how to set up a virtual machine on the Google Cloud Platform (GCP) to deploy your pre-configured Corda node and automatically connect to Testnet.

#### Pre-requisites

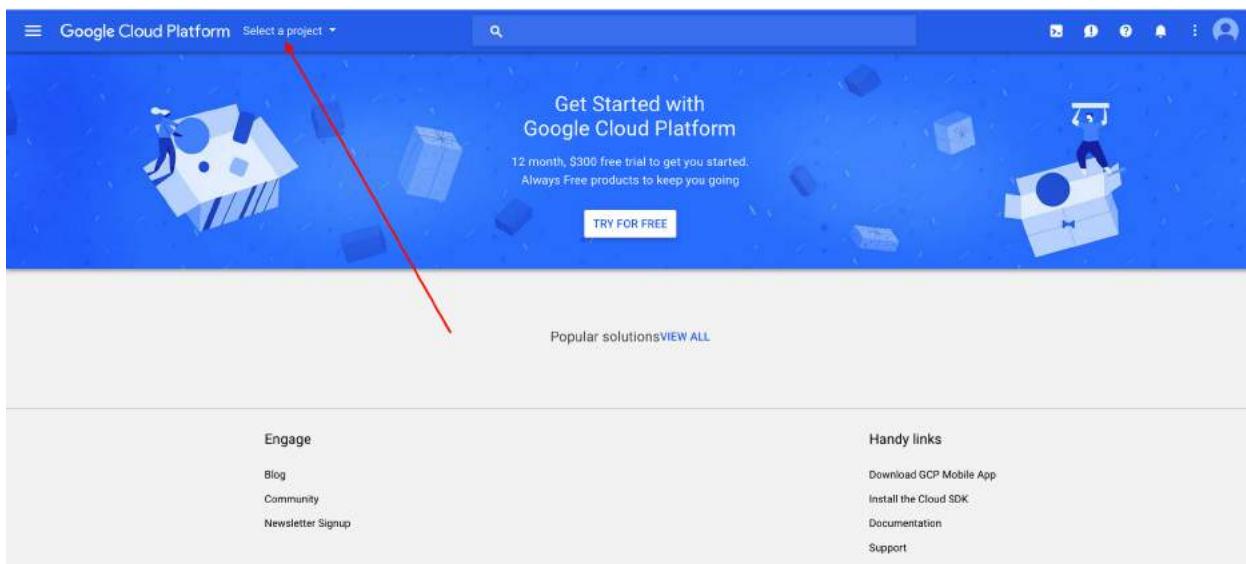
- Ensure you have a registered Google Cloud Platform account with billing enabled (<https://cloud.google.com/billing/docs/how-to/manage-billing-account>) which can create virtual machines under your subscription(s) and you are logged on to the GCP console: <https://console.cloud.google.com>.

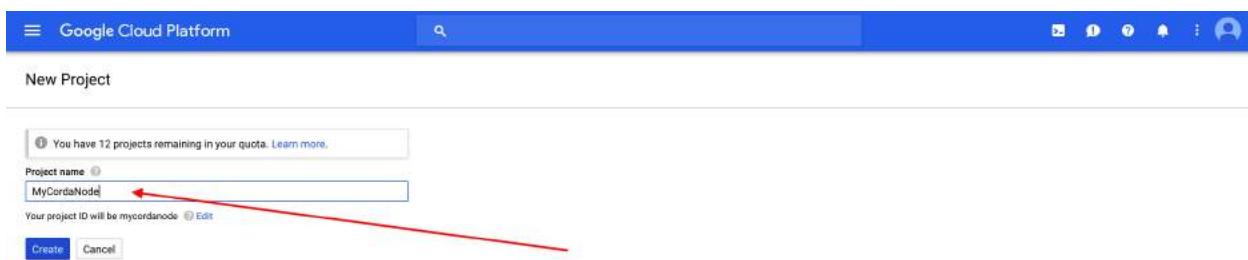
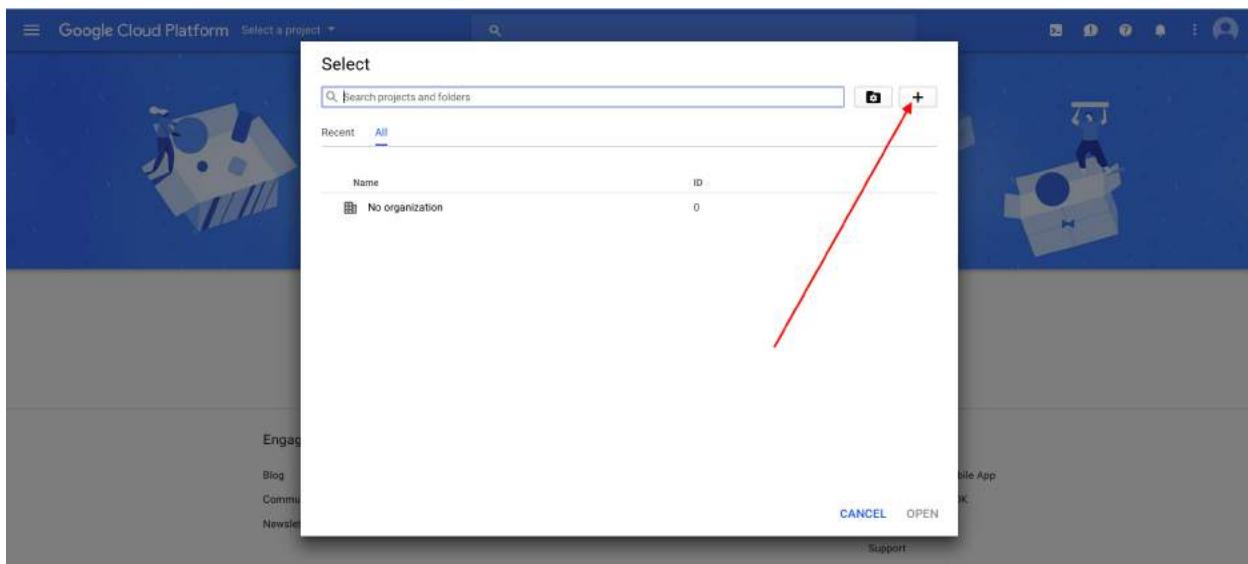
#### Deploy Corda node

Browse to <https://console.cloud.google.com> and log in with your Google credentials.

##### STEP 1: Create a GCP Project

In the project drop down click on the plus icon to create a new project to house your Corda resources.

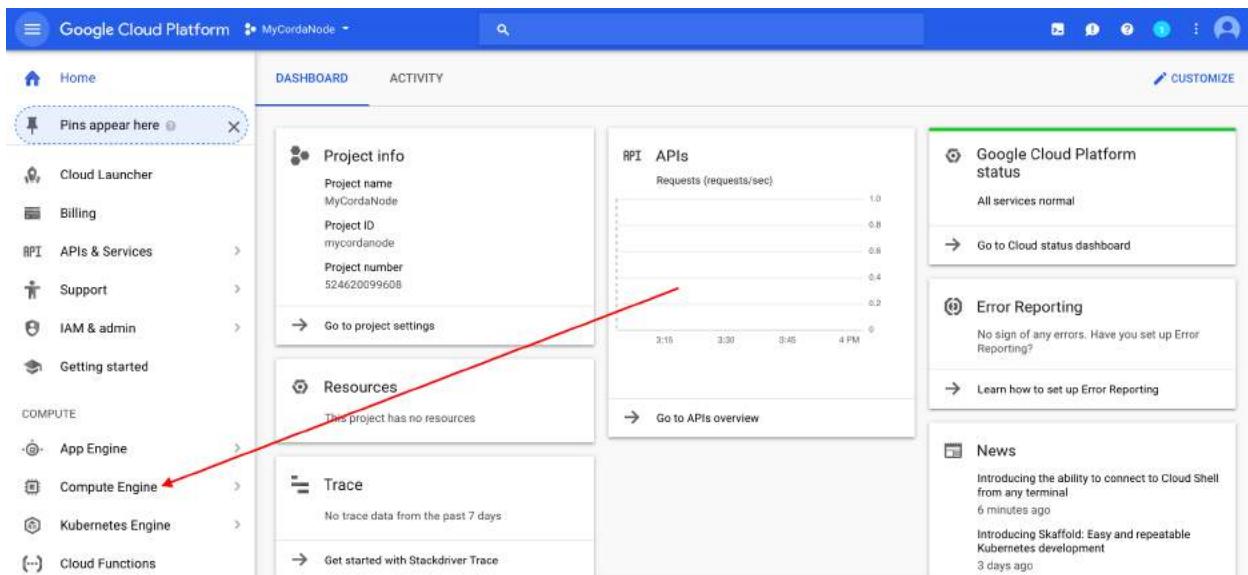




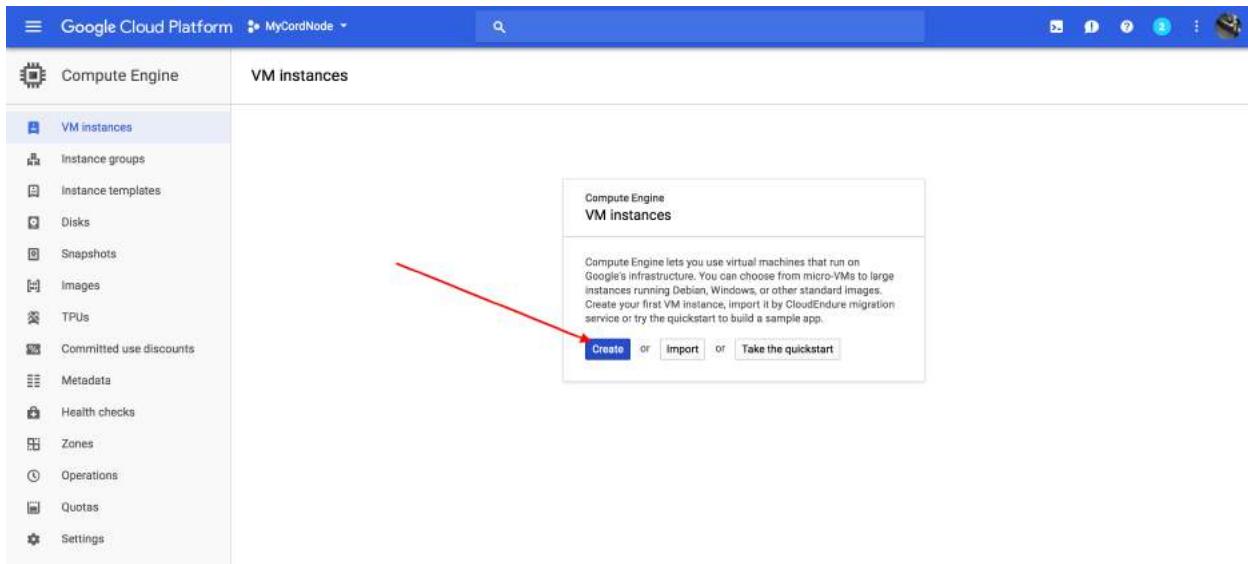
Enter a project name and click Create.

### STEP 2: Launch the VM

In the left hand side nav click on Compute Engine.



Click on Create Instance.

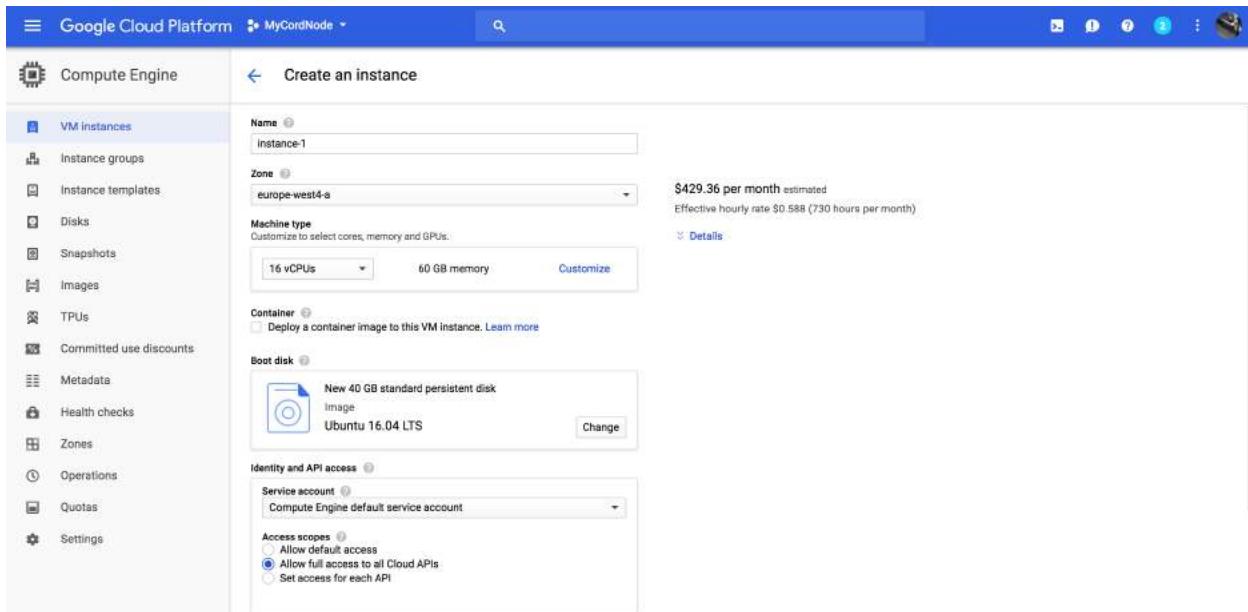


Fill in the form with the desired VM specs:

Recommended minimum 4vCPU with 15GB memory and 40GB Persistent disk. Ubuntu 16.04 LTS.

Allow full API access.

Dont worry about firewall settings as you will configure those later.



Click Create and wait a few sections for your instance to provision and start running.

### STEP 3: Connect to your VM and set up the environment

Once your instance is running click on the SSH button to launch a cloud SSH terminal in a new window.

Secure | https://ssh.cloud.google.com/projects/mycordnode/zones/europe-west4-a/instances/instance-1?authuser=0&hl=en\_US&pr...

```
Connected, host fingerprint: ssh-rsa 2048 DE:73:8D:E0:10:63:43:E8:C2:A9:BD:8A:3A:51:10:F4:7A:60:BE:D6:B1:A0:A8:90:E
:AC:2D:54:AF:1F:69:6D
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.13.0-1011-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
 http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

nick_arini@instance-1:~$
```

Run the following to configure the firewall to allow Corda traffic

```
gcloud compute firewall-rules create nodetonode --allow tcp:10002
gcloud compute firewall-rules create nodetorpc --allow tcp:10003
gcloud compute firewall-rules create webserver --allow tcp:8080
```

Promote the ephemeral IP address associated with this instance to a static IP address.

First check the region and select the one you are using from the list:

```
gcloud compute regions list
```

Find your external IP:

```
gcloud compute addresses list
```

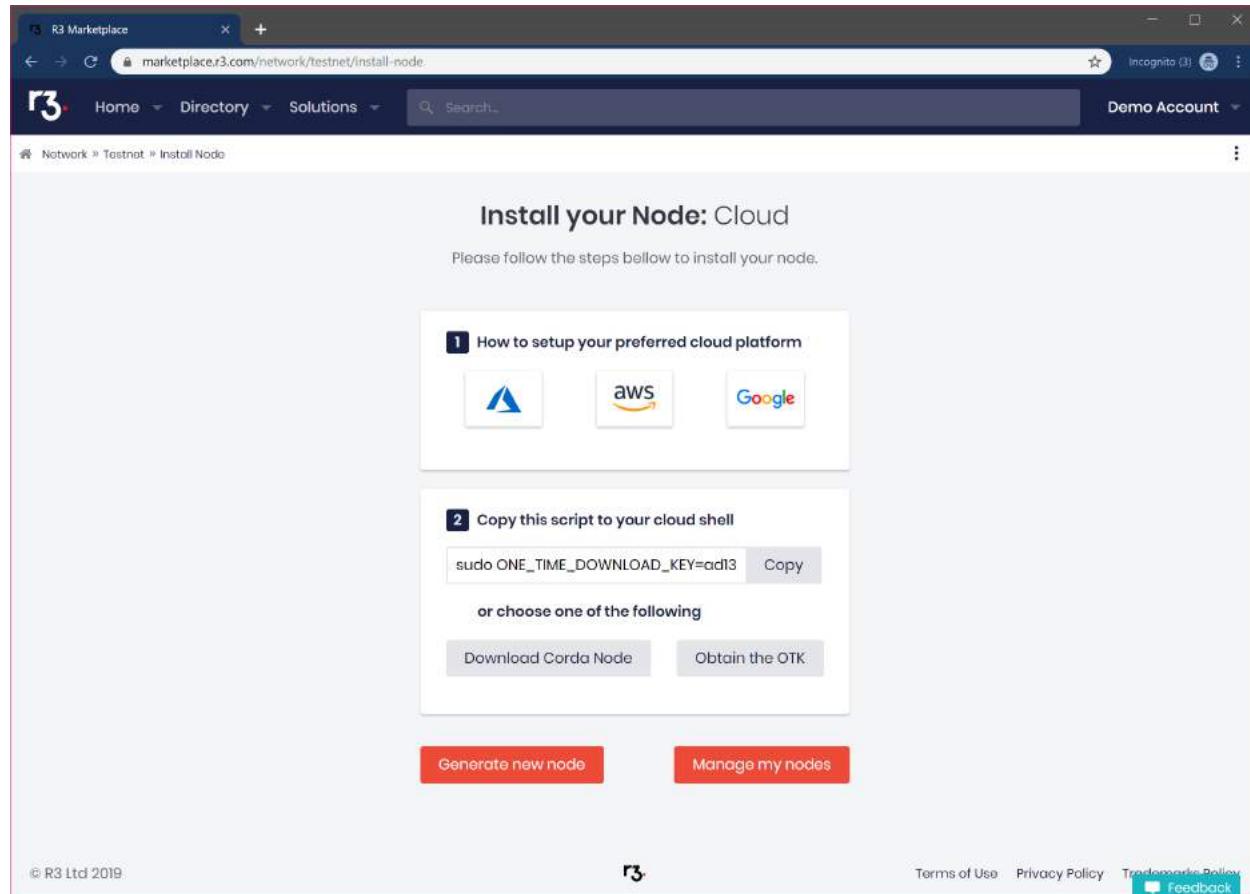
Run this command with the ephemeral IP address as the argument to the `--addresses` flag and the region:

```
gcloud compute addresses create corda-node --addresses 35.204.53.61 --region europe-west4
```

#### STEP 4: Download and set up your Corda node

Now your GCP environment is configured you can switch to the Testnet web application and click on the copy to clipboard button to get a one time installation script.

**Note:** If you have not already set up your account on Testnet then please visit <https://marketplace.r3.com/network/testnet> and sign up.



You can generate as many Testnet identities as you like by refreshing this page to generate a new one time link.

In the terminal of your cloud instance paste the command you just copied to install and run your unique Corda instance:

```
sudo ONE_TIME_DOWNLOAD_KEY=YOUR_UNIQUE_DOWNLOAD_KEY_HERE bash -c "$(curl -L https://onboarder.prod.ws.r3.com/api/user/node/TESTNET/install.sh)"
```

**Warning:** This command will execute the install script as ROOT on your cloud instance. You may wish to examine the script prior to executing it on your machine.

You can follow the progress of the installation by typing the following command in your terminal:

```
tail -f /opt/corda/logs/node-<VM-NAME>.log
```

## Testing your deployment

To test your deployment is working correctly follow the instructions in [Using the Node Explorer to test a Corda node on Corda Testnet](#) to set up the Finance CorDapp and issue cash to a counterparty.

This will also demonstrate how to install a custom CorDapp.

### 17.7.4 Deploying Corda to Corda Testnet from your local environment

#### Contents

- *Deploying Corda to Corda Testnet from your local environment*
  - *Pre-requisites*
  - *Set up your local network*
    - \* *Set up static IP address local host machine*
    - \* *Set up port forwarding on your router*
    - \* *Open firewall ports*
    - \* *Optional: Configure a static external IP address*
    - \* *Test if the ports are open*
    - \* *Download and install your node*
  - *Testing your deployment*

This document explains how to set up your local network to enable a Corda node to connect to the Corda Testnet. This assumes you are downloading a node ZIP from: <https://marketplace.r3.com/network/testnet>.

#### Pre-requisites

- Register for an account on <https://marketplace.r3.com/>.

#### Set up your local network

For a Corda node to be able to connect to the Corda Testnet and be reachable by counterparties on that network it needs to be reachable on the open internet. Corda is a server which requires an externally visible IP address and several ports in order to operate correctly.

We recommend running your Corda node on cloud infrastructure. If you wish to run Corda on your local machine then you will need to configure your network to enable the Corda node to be reachable from the internet.

---

**Note:** You will need access to your network router/gateway to the internet. If you do not have direct access then contact your administrator.

---

The following steps will describe how to use port forwarding on your router to make sure packets intended for Corda are routed to the right place on your local network.

### Set up static IP address local host machine

The next steps will configure your router to forward packets to the Corda node, but for this it is required to set the host machine to have a static IP address. If this isn't done, and the network is using DHCP dynamic address allocation then the next time the host machine is rebooted it may be on a different IP and the port forwarding will no longer work.

Please consult your operating system documentation for instructions on setting a static IP on the host machine.

### Set up port forwarding on your router

Port forwarding is a method of making a computer on your network accessible to computers on the Internet, even though it is behind a router.

---

**Note:** All routers are slightly different and you will need to consult the documentation for your specific make and model.

---

Log in to the admin page of your router (often 192.168.0.1) in your browser bar.

---

**Note:** Router administration IP and log in credentials are usually on the bottom or side of your router.

---

Navigate to the port forwarding section of the admin console.

Add rules for the following ports which Corda uses:

```
10002  
10003  
8080
```

---

**Note:** These ports are the defaults for Testnet which are specified in the node.conf. If these conflict with existing services on your host machine they can be changed in the /opt/corda/node.conf file.

---

For each rule you will also typically have to specify the rule name, the static IP address of the host machine we configured earlier (the same in each case) and the protocol (which is TCP in all cases here).

Please consult your router documentation for specific details on enabling port forwarding.

### Open firewall ports

If you are operating a firewall on your host machine or local network you will also need to open the above ports for incoming traffic.

Please consult your firewall documentation for details.

### Optional: Configure a static external IP address

Corda expects nodes to have stable addresses over long periods of time. ISPs typically assign dynamic IP addresses to a router and so if your router is rebooted it may not obtain the same external IP and therefore your Corda node will change its address on the Testnet.

You can request a static IP address from your ISP however this may incur a cost.

If the IP address does change then this doesn't cause issues but it will result in an update to the network map which then needs to be propagated to all peers in the network. There may be some delay in the ability to transact while this happens.

**Warning:** Corda nodes are expected to be online all the time and will send a heartbeat to the network map server to indicate they are operational. If they go offline for a period of time (~24 hours in the case of Testnet) then the node will be removed from the network map. Any nodes which have queued messages for your node will drop these messages, they won't be delivered and unexpected behaviour may occur.

### Test if the ports are open

You can use a port checking tool to make sure the ports are open properly.

### Download and install your node

Navigate to <https://marketplace.r3.com/network/testnet/install-node>.

Click on the Download Corda Node button and wait for the ZIP file to download:

**Install your Node: Cloud**

Please follow the steps below to install your node.

**1 How to setup your preferred cloud platform**

**2 Copy this script to your cloud shell**

`sudo ONE_TIME_DOWNLOAD_KEY=ad13` [Copy](#)

or choose one of the following

[Download Corda Node](#) [Obtain the OTK](#)

[Generate new node](#) [Manage my nodes](#)

© R3 Ltd 2019 Terms of Use Privacy Policy Trademarks Policy Feedback

Unzip the file in your Corda root directory:

```
mkdir corda
cd corda
cp <PATH_TO_DOWNLOAD>/node.zip .
unzip node.zip
cd node
```

Run the `run-corda.sh` script to start your Corda node.

```
./run-corda.sh
```

Congratulations! You now have a running Corda node on Testnet.

**Warning:** It is possible to copy the `node.zip` file from your local machine to any other host machine and run the Corda node from there. Do not run multiple copies of the same node (i.e. with the same identity). If a new copy of the node appears on the network then the network map server will interpret this as a change in the address of the node and route traffic to the most recent instance. Any states which are on the old node will no longer be available and undefined behaviour may result. Please provision a new node from the application instead.

## Testing your deployment

To test your deployment is working correctly follow the instructions in [Using the Node Explorer to test a Corda node on Corda Testnet](#) to set up the Finance CorDapp and issue cash to a counterparty.

## 17.8 Using the Node Explorer to test a Corda node on Corda Testnet

This document will explain how to test the installation of a Corda node on Testnet.

### 17.8.1 Prerequisites

This guide assumes you have deployed a Corda node to the Corda Testnet.

---

**Note:** If you need to set up a node on Testnet first please follow the instructions: [Joining Corda Testnet](#).

---

### 17.8.2 Get the testing tools

To run the tests and make sure your node is connecting correctly to the network you will need to download and install a couple of resources.

1. Log into your Cloud VM via SSH.
2. Stop the Corda node(s) running on your cloud instance.

```
ps aux | grep corda.jar | awk '{ print $2 }' | xargs sudo kill
```

3. Download the finance CorDapp

In the terminal on your cloud instance run:

```
wget https://ci-artifactory.corda.r3cev.com/artifactory/corda-releases/net/corda/
    ↪corda-finance-contracts/4.1/corda-finance-contracts-4.1.jar
wget https://ci-artifactory.corda.r3cev.com/artifactory/corda-releases/net/corda/
    ↪corda-finance-workflows/4.1/corda-finance-workflows-4.1.jar
```

This is required to run some flows to check your connections, and to issue/transfer cash to counterparties. Copy it to the Corda installation location:

```
sudo cp /home/<USER>/corda-finance-4.1-corda.jar /opt/corda/cordapps/
```

4. Run the following to create a config file for the finance CorDapp:

```
echo "issuableCurrencies = [ USD ]" > /opt/corda/cordapps/config/corda-finance-4.
    ↪1-corda.conf
```

5. Restart the Corda node:

```
cd /opt/corda
sudo ./run-corda.sh
```

Your node is now running the finance Cordapp.

---

**Note:** You can double-check that the CorDapp is loaded in the log file `/opt/corda/logs/node-<VM-NAME>.log`. This file will list installed apps at startup. Search for `Loaded CorDapps` in the logs.

---

- Now download the Node Explorer to your **LOCAL** machine:

**Note:** Node Explorer is a JavaFX GUI which connects to the node over the RPC interface and allows you to send transactions.

Download the Node Explorer from here:

```
http://ci-artifactory.corda.r3cev.com/artifactory/corda-releases/net/corda/corda-
tools-explorer/4.1-corda/corda-tools-explorer-4.1-corda.jar
```

**Warning:** This Node Explorer is incompatible with the Corda Enterprise distribution and vice versa as they currently use different serialisation schemes (Kryo vs AMQP).

- Run the Node Explorer tool on your **LOCAL** machine.

```
java -jar corda-tools-explorer-4.1-corda.jar
```



### 17.8.3 Connect to the node

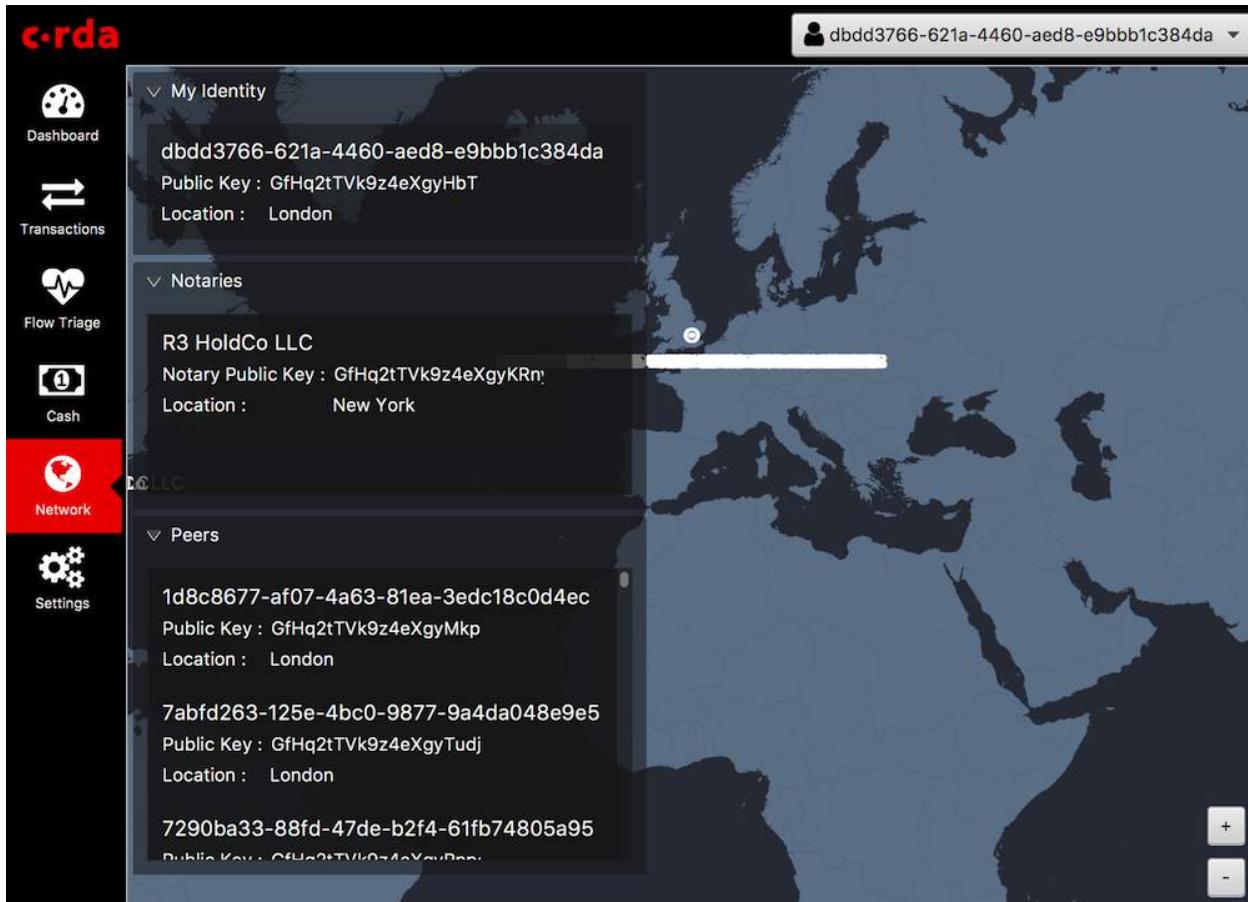
To connect to the node you will need:

- The IP address of your node (the public IP of your cloud instance). You can find this in the instance page of your cloud console.
- The port number of the RPC interface to the node, specified in `/opt/corda/node.conf` in the `rpcSettings` section, (by default this is 10003 on Testnet).
- The username and password of the RPC interface of the node, also in the `node.conf` in the `rpcUsers` section, (by default the username is `cordazoneservice` on Testnet).

Click on **Connect** to log into the node.

### 17.8.4 Check your network identity and counterparties

Once Explorer has logged in to your node over RPC click on the Network tab in the side navigation of the Explorer UI:



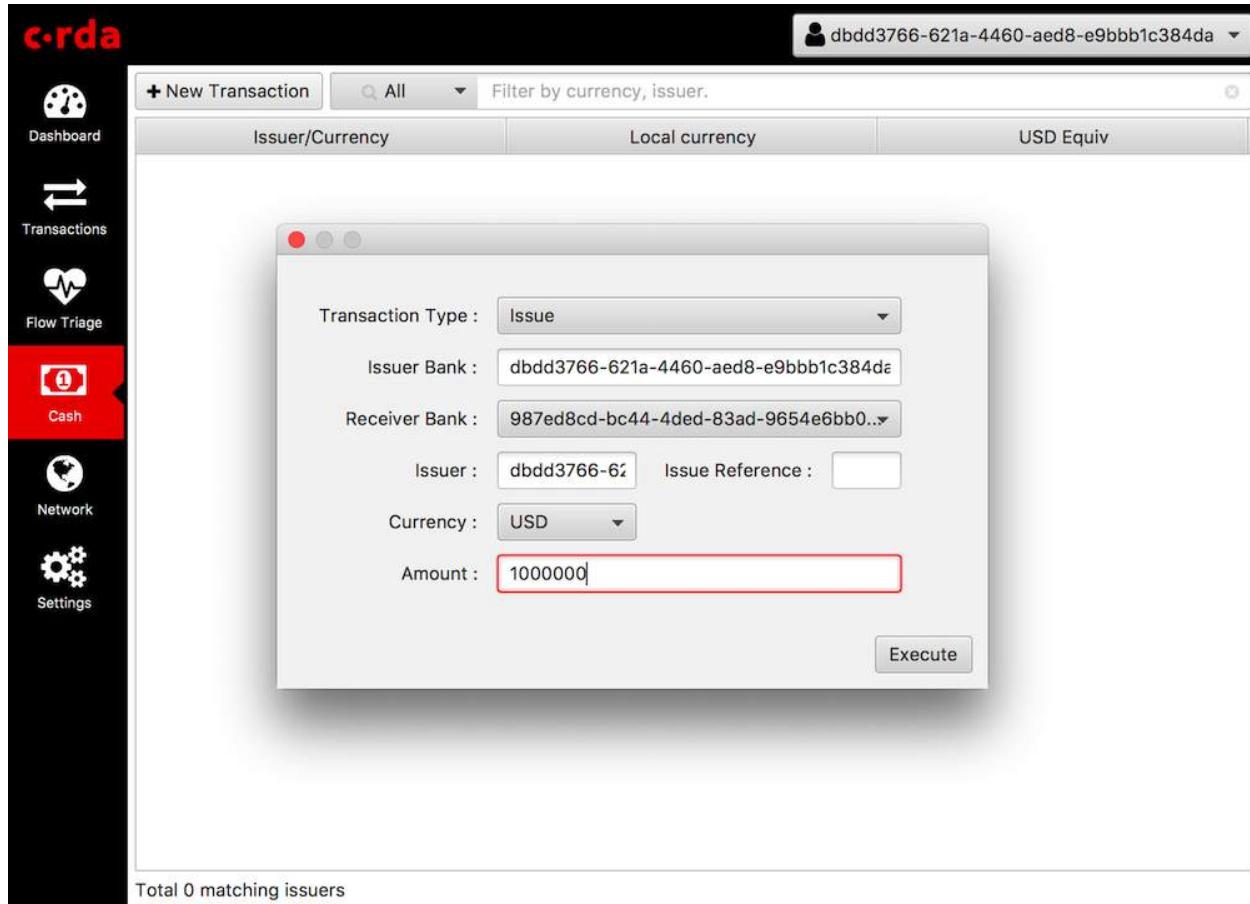
If your Corda node is correctly configured and connected to the Testnet then you should be able to see the identities of your node, the Testnet notary and the network map listing all the counterparties currently on the network.

### 17.8.5 Test issuance transaction

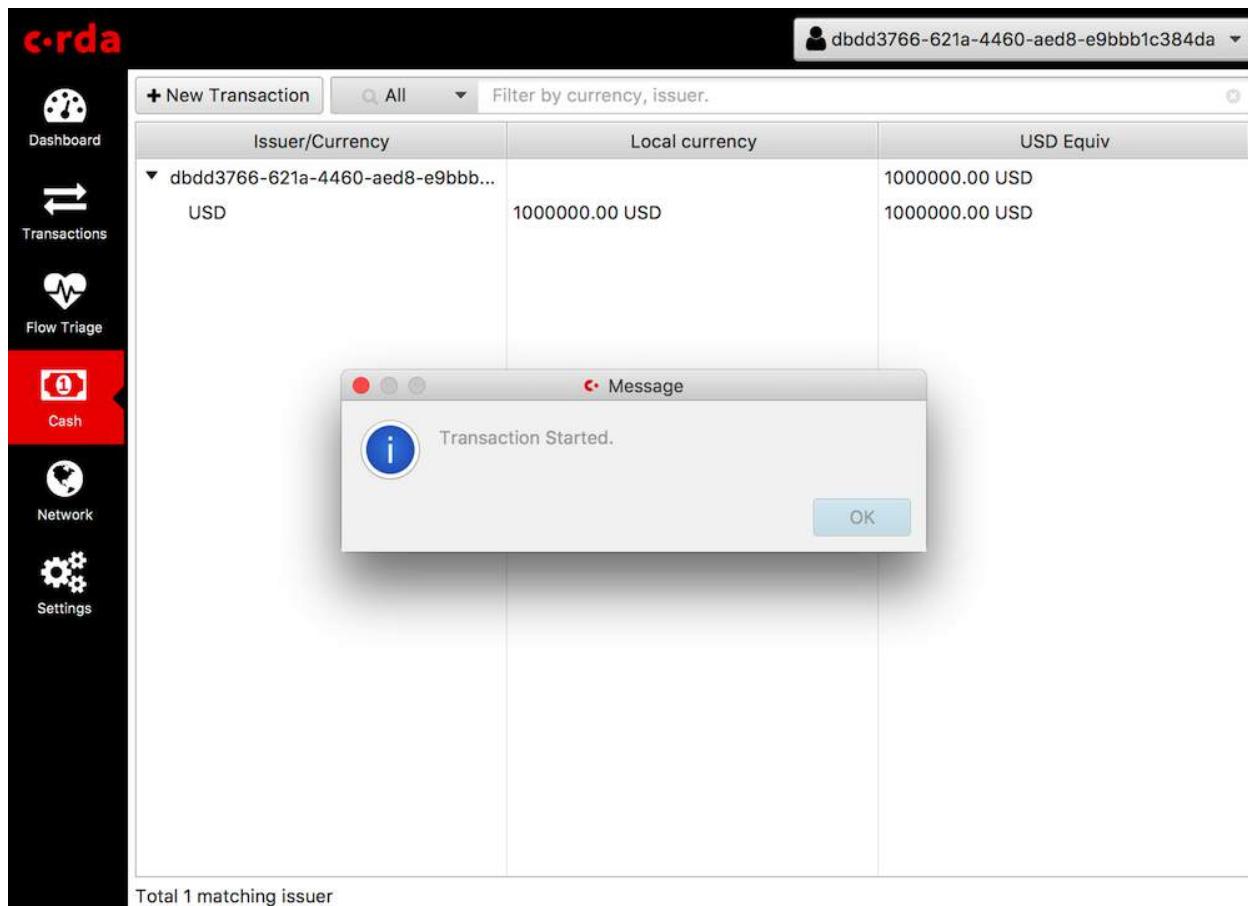
Now we are going to try and issue some cash to a ‘bank’. Click on the Cash tab.

The screenshot shows the Corda Node Explorer interface. On the left is a sidebar with icons for Dashboard, Transactions, Flow Triage, Cash (which is selected and highlighted in red), Network, and Settings. The main area has a header with a user icon and the ID dbdd3766-621a-4460-aed8-e9bbb1c384da. Below the header is a search bar with 'All' and a filter dropdown set to 'Filter by currency, issuer.' There are three columns in a table: 'Issuer/Currency', 'Local currency', and 'USD Equiv'. A message 'No content in table' is displayed. At the bottom of the main area, it says 'Total 0 matching issuers'.

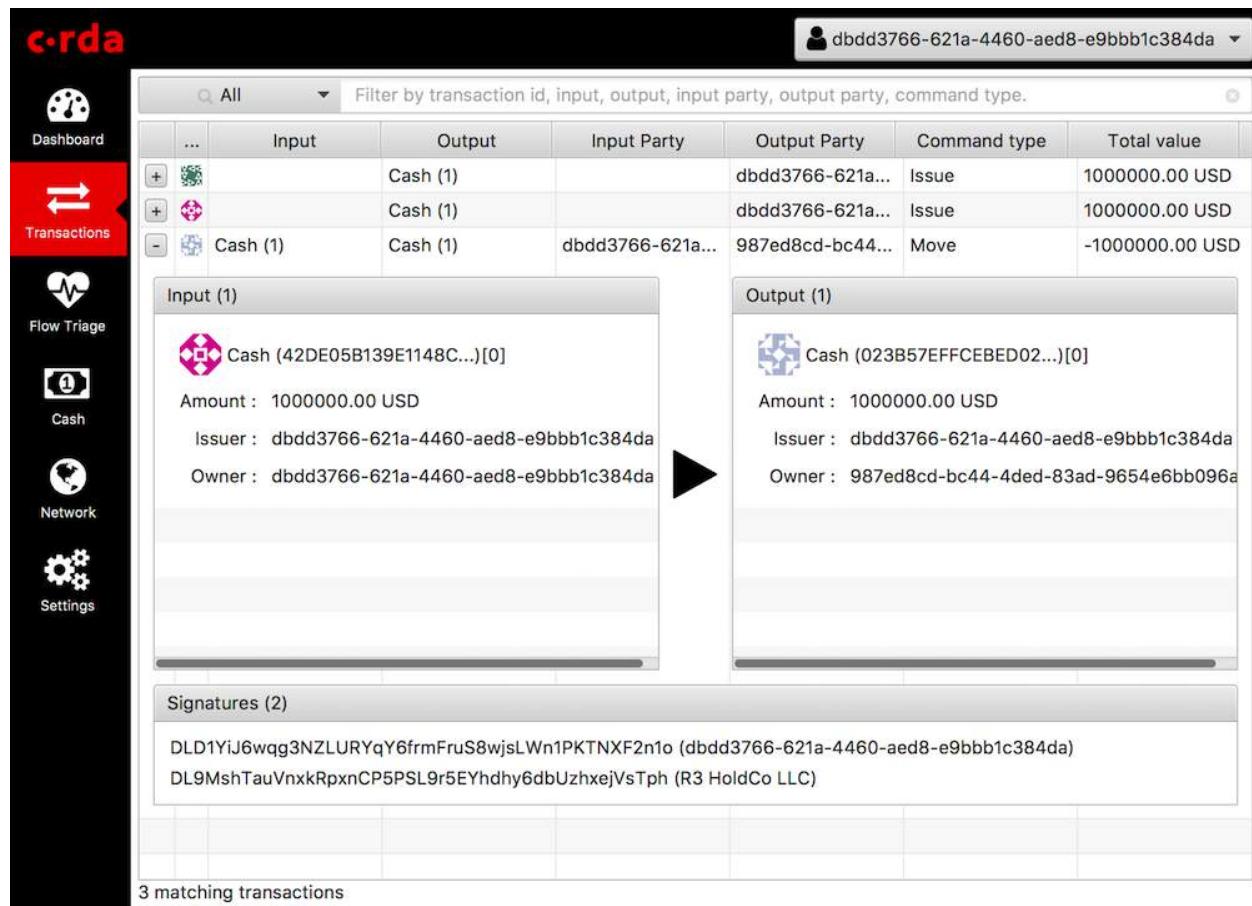
Now click on New Transaction and create an issuance to a known counterparty on the network by filling in the form:



Click Execute and the transaction will start.



Click on the red X to close the notification window and click on Transactions tab to see the transaction in progress, or wait for a success message to be displayed:



Congratulations! You have now successfully installed a CorDapp and executed a transaction on the Corda Testnet.

## 17.9 Setting up a dynamic compatibility zone

### Contents

- *Setting up a dynamic compatibility zone*
  - *Do you need to create your own dynamic compatibility zone?*
  - *Why create your own zone?*
  - *How to create your own compatibility zone*
    - \* *Using an existing network map implementation*
    - \* *Creating your own network map implementation*
      - *Writing a network map server*
      - *Writing a doorman server*
      - *Setting zone parameters*
    - \* *Selecting parameter values*

### 17.9.1 Do you need to create your own dynamic compatibility zone?

By *dynamic compatibility zone*, we mean a compatibility zone that relies on a network map server to allow nodes to join dynamically, instead of requiring each node to be bootstrapped and have the node-infos distributed manually. While this may sound appealing, think twice before going down this route:

1. If you need to test a CorDapp, it is easier to create a test network using the network bootstrapper tool (see below)
2. If you need to control who uses your CorDapp, it is easier to apply permissioning by creating a business network (see below)

**Testing.** Creating a production-ready zone isn't necessary for testing as you can use the *network bootstrapper* tool to create all the certificates, keys, and distribute the needed map files to run many nodes. The bootstrapper can create a network locally on your desktop/laptop but it also knows how to automate cloud providers via their APIs and using Docker. In this way you can bring up a simulation of a real Corda network with different nodes on different machines in the cloud for your own testing. Testing this way has several advantages, most obviously that you avoid race conditions in your tests caused by nodes/tests starting before all map data has propagated to all nodes. You can read more about the reasons for the creation of the bootstrapper tool [in a blog post on the design thinking behind Corda's network map infrastructure](#).

**Permissioning.** And creating a zone is also unnecessary for imposing permissioning requirements beyond that of the base Corda network. You can control who can use your app by creating a *business network*. A business network is what we call a coalition of nodes that have chosen to run a particular app within a given commercial context. Business networks aren't represented in the Corda API at this time, partly because the technical side is so simple. You can create one via a simple three step process:

1. Distribute a list of X.500 names that are members of your business network. You can use the reference [Business Network Membership Service implementation](#). Alternatively, you could do this is by hosting a text file with one name per line on your website at a fixed HTTPS URL. You could also write a simple request/response flow that serves the list over the Corda protocol itself, although this requires the business network to have its own node.
2. Write a bit of code that downloads and caches the contents of this file on disk, and which loads it into memory in the node. A good place to do this is in a class annotated with `@CordaService`, because this class can expose a `Set<Party>` field representing the membership of your service.
3. In your flows use `serviceHub.findService` to get a reference to your `@CordaService` class, read the list of members and at the start of each flow, throw a `FlowException` if the counterparty isn't in the membership list.

In this way you can impose a centrally controlled ACL that all members will collectively enforce.

---

**Note:** A production-ready Corda network and a new iteration of the testnet will be available soon.

---

### 17.9.2 Why create your own zone?

The primary reason to create a zone and provide the associated infrastructure is control over *network parameters*. These are settings that control Corda's operation, and on which all users in a network must agree. Failure to agree would create the Corda equivalent of a blockchain "hard fork". Parameters control things like the root of identity, how quickly users should upgrade, how long nodes can be offline before they are evicted from the system and so on.

Creating a zone involves the following steps:

1. Create the zone private keys and certificates. This procedure is conventional and no special knowledge is required: any self-signed set of certificates can be used. A professional quality zone will probably keep the keys inside a hardware security module (as the main Corda network and test networks do).
2. Write a network map server.

3. Optionally, create a doorman server.
4. Finally, you would select and generate your network parameter file.

### 17.9.3 How to create your own compatibility zone

#### Using an existing network map implementation

You can use an existing network map implementation such as the Cordite Network Map Service to create a dynamic compatibility zone.

#### Creating your own network map implementation

##### Writing a network map server

This server implements a simple HTTP based protocol described in the “[The network map](#)” page. The map server is responsible for gathering NodeInfo files from nodes, storing them, and distributing them back to the nodes in the zone. By doing this it is also responsible for choosing who is in and who is out: having a signed identity certificate is not enough to be a part of a Corda zone, you also need to be listed in the network map. It can be thought of as a DNS equivalent. If you want to de-list a user, you would do it here.

It is very likely that your map server won’t be entirely standalone, but rather, integrated with whatever your master user database is.

The network map server also distributes signed network parameter files and controls the rollout schedule for when they become available for download and opt-in, and when they become enforced. This is again a policy decision you will probably choose to place some simple UI or workflow tooling around, in particular to enforce restrictions on who can edit the map or the parameters.

##### Writing a doorman server

This step is optional because your users can obtain a signed certificate in many different ways. The doorman protocol is again a very simple HTTP based approach in which a node creates keys and requests a certificate, polling until it gets back what it expects. However, you could also integrate this process with the rest of your signup process. For example, by building a tool that’s integrated with your payment flow (if payment is required to take part in your zone at all). Alternatively you may wish to distribute USB smartcard tokens that generate the private key on first use, as is typically seen in national PKIs. There are many options.

If you do choose to make a doorman server, the bulk of the code you write will be workflow related. For instance, related to keeping track of an applicant as they proceed through approval. You should also impose any naming policies you have in the doorman process. If names are meant to match identities registered in government databases then that should be enforced here, alternatively, if names can be self-selected or anonymous, you would only bother with a deduplication check. Again it will likely be integrated with a master user database.

Corda does not currently provide a doorman or network map service out of the box, partly because when stripped of the zone specific policy there isn’t much to them: just a basic HTTP server that most programmers will have favourite frameworks for anyway.

The protocol is:

- If \$URL = `https://some.server.com/some/path`
- Node submits a PKCS#10 certificate signing request using HTTP POST to `$URL/certificate`. It will have a MIME type of `application/octet-stream`. The `Client-Version` header is set to be “1.0”.

- The server returns an opaque string that references this request (let's call it `$requestid`, or an HTTP error if something went wrong).
- The returned request ID should be persisted to disk, to handle zones where approval may take a long time due to manual intervention being required.
- The node starts polling `$URL/$requestid` using HTTP GET. The poll interval can be controlled by the server returning a response with a `Cache-Control` header.
- If the request is answered with a `200 OK` response, the body is expected to be a zip file. Each file is expected to be a binary X.509 certificate, and the certs are expected to be in order.
- If the request is answered with a `204 No Content` response, the node will try again later.
- If the request is answered with a `403 Not Authorized` response, the node will treat that as request rejection and give up.
- Other response codes will cause the node to abort with an exception.

## Setting zone parameters

Zone parameters are stored in a file containing a Corda AMQP serialised `SignedDataWithCert<NetworkParameters>` object. It is easy to create such a file with a small Java or Kotlin program. The `NetworkParameters` object is a simple data holder that could be read from e.g. a config file, or settings from a database. Signing and saving the resulting file is just a few lines of code. A full example can be found in `NetworkParametersCopier.kt` in the source tree, but a flavour of it looks like this:

```
NetworkParameters networkParameters = new NetworkParameters(
    4,                                     // minPlatformVersion
    Collections.emptyList(),   // the `NotaryInfo`s of all the network's notaries
    1024 * 1024 * 20,        // maxMessageSize
    1024 * 1024 * 15,        // maxTransactionSize
    Instant.now(),           // modifiedTime
    2,                         // epoch
    Collections.emptyMap()   // whitelisted contract code JARs
);
CertificateAndKeyPair signingCertAndKeyPair = loadNetworkMapCA();
SerializedBytes<SignedDataWithCert<NetworkParameters>> bytes = SerializedBytes.
    ↪from(netMapCA.sign(networkParameters));
Files.copy(bytes.open(), Paths.get("params-file"));
```

```
val networkParameters = NetworkParameters(
    minimumPlatformVersion = 4,
    notaries = listOf(...),
    maxMessageSize = 1024 * 1024 * 20    // 20mb, for example.
    maxTransactionSize = 1024 * 1024 * 15,
    modifiedTime = Instant.now(),
    epoch = 2,
    ... etc ...
)
val signingCertAndKeyPair: CertificateAndKeyPair = loadNetworkMapCA()
val signedParams: SerializedBytes<SignedNetworkParameters> = signingCertAndKeyPair.
    ↪sign(networkParameters).serialize()
signedParams.open().copyTo(Paths.get("/some/path"))
```

Each individual parameter is documented in [the JavaDocs/KDocs for the `NetworkParameters` class](#). The network map certificate is usually chained off the root certificate, and can be created according to the instructions above. Each time the zone parameters are changed, the epoch should be incremented. Epochs are essentially version numbers for the

parameters, and they therefore cannot go backwards. Once saved, the new parameters can be served by the network map server.

### Selecting parameter values

How to choose the parameters? This is the most complex question facing you as a new zone operator. Some settings may seem straightforward and others may involve cost/benefit tradeoffs specific to your business. For example, you could choose to run a validating notary yourself, in which case you would (in the absence of SGX) see all the users' data. Or you could run a non-validating notary, with BFT fault tolerance, which implies recruiting others to take part in the cluster.

New network parameters will be added over time as Corda evolves. You will need to ensure that when your users upgrade, all the new network parameters are being served. You can ask for advice on the [corda-dev mailing list](#).

## 17.10 Setting up a notary service

Corda comes with several notary implementations built-in:

1. **Single-node**: a simple notary service that persists notarisation requests in the node's database. It is easy to set up and is recommended for testing, and production networks that do not have strict availability requirements.
2. **Crash fault-tolerant (experimental)**: a highly available notary service operated by a single party.
3. **Byzantine fault-tolerant (experimental)**: a decentralised highly available notary service operated by a group of parties.

### 17.10.1 Single-node

To have a regular Corda node provide a notary service you simply need to set appropriate notary configuration values before starting it:

```
notary : { validating : false }
```

For a validating notary service specify:

```
notary : { validating : true }
```

See [Validation](#) for more details about validating versus non-validating notaries.

For clients to be able to use the notary service, its identity must be added to the network parameters. This will be done automatically when creating the network, if using [Network Bootstrapper](#). See [Networks](#) for more details.

### 17.10.2 Crash fault-tolerant (experimental)

Corda provides a prototype [Raft-based](#) highly available notary implementation. You can try it out on our [notary demo](#) page. Note that it has known limitations and is not recommended for production use.

### 17.10.3 Byzantine fault-tolerant (experimental)

A prototype BFT notary implementation based on [BFT-Smart](#) is available. You can try it out on our [notary demo](#) page. Note that it is still experimental and there is active work ongoing for a production ready solution. Additionally,

BFT-Smart requires Java serialization which is disabled by default in Corda due to security risks, and it will only work in dev mode where this can be customised.

We do not recommend using it in any long-running test or production deployments.

## OFFICIAL CORDA DOCKER IMAGE

### 18.1 Running a node connected to a Compatibility Zone in Docker

---

**Note:** Requirements: A valid node.conf and a valid set of certificates - (signed by the CZ)

---

In this example, the certificates are stored at /home/user/cordaBase/certificates, the node configuration is in /home/user/cordaBase/config/node.conf and the CorDapps to run are in /path/to/cordapps

```
docker run -ti \
    --memory=2048m \
    --cpus=2 \
    -v /home/user/cordaBase/config:/etc/corda \
    -v /home/user/cordaBase/certificates:/opt/corda/certificates \
    -v /home/user/cordaBase/persistence:/opt/corda/persistence \
    -v /home/user/cordaBase/logs:/opt/corda/logs \
    -v /path/to/cordapps:/opt/corda/cordapps \
    -p 10200:10200 \
    -p 10201:10201 \
    corda/corda-zulu-5.0-snapshot:latest
```

As the node runs within a container, several mount points are required:

1. CorDapps - CorDapps must be mounted at location /opt/corda/cordapps
2. Certificates - certificates must be mounted at location /opt/corda/certificates
3. Config - the node config must be mounted at location /etc/corda/node.config
4. Logging - all log files will be written to location /opt/corda/logs

If using the H2 database:

5. Persistence - the folder to hold the H2 database files must be mounted at location /opt/corda/persistence

---

**Note:** If there is no dataSourceProperties key in the node.conf, the docker container overrides the url for H2 to point to the persistence directory by default so that the database can be accessed outside the container

---

### 18.2 Running a node connected to a Bootstrapped Network

---

**Note:** Requirements: A valid node.conf, a valid set of certificates, and an existing network-parameters file

---

In this example, we have previously generated a network-parameters file using the bootstrapper tool, which is stored at /home/user/sharedFolder/network-parameters

```
docker run -ti \
    --memory=2048m \
    --cpus=2 \
    -v /home/user/cordaBase/config:/etc/corda \
    -v /home/user/cordaBase/certificates:/opt/corda/certificates \
    -v /home/user/cordaBase/persistence:/opt/corda/persistence \
    -v /home/user/cordaBase/logs:/opt/corda/logs \
    -v /home/TeamCityOutput/cordapps:/opt/corda/cordapps \
    -v /home/user/sharedFolder/node-infos:/opt/corda/additional-node-infos \
    -v /home/user/sharedFolder/network-parameters:/opt/corda/network-parameters \
    -p 10200:10200 \
    -p 10201:10201 \
    corda/corda-zulu-5.0-snapshot:latest
```

There is a new mount /home/user/sharedFolder/node-infos:/opt/corda/additional-node-infos which is used to hold the nodeInfo of all the nodes within the network. As the node within the container starts up, it will place its own nodeInfo into this directory. This will allow other nodes also using this folder to see this new node.

## 18.3 Generating configs and certificates

It is possible to utilize the image to automatically generate a sensible minimal configuration for joining an existing Corda network.

## 18.4 Joining TestNet

---

**Note:** Requirements: A valid registration for TestNet and a one-time code for joining TestNet. Certificate and configuration folders should be accessible from the container. Docker will create folders using the permissions of its daemon if they don't exist and the container may fail accessing them.

---

```
docker run -ti \
    -e MY_PUBLIC_ADDRESS="corda-node.example.com" \
    -e ONE_TIME_DOWNLOAD_KEY="bbcb189e-9e4f-4b27-96db-134e8f592785" \
    -e LOCALITY="London" -e COUNTRY="GB" \
    -v /home/user/docker/config:/etc/corda \
    -v /home/user/docker/certificates:/opt/corda/certificates \
    corda/corda-zulu-5.0-snapshot:latest config-generator --testnet
```

\$MY\_PUBLIC\_ADDRESS will be the public address that this node will be advertised on. \$ONE\_TIME\_DOWNLOAD\_KEY is the one-time code provided for joining TestNet. \$LOCALITY and \$COUNTRY must be set to the values provided when joining TestNet.

When the container has finished executing config-generator the following will be true

1. A skeleton, but sensible minimum node.conf is present in /home/user/docker/config

2. A set of certificates signed by TestNet in /home/user/docker/certificates

It is now possible to start the node using the generated config and certificates

```
docker run -ti \
--memory=2048m \
--cpus=2 \
-v /home/user/docker/config:/etc/corda \
-v /home/user/docker/certificates:/opt/corda/certificates \
-v /home/user/docker/persistence:/opt/corda/persistence \
-v /home/user/docker/logs:/opt/corda/logs \
-v /home/user/corda/samples/bank-of-corda-demo/build/nodes/BankOfCorda/
-cordapps:/opt/corda/cordapps \
-p 10200:10200 \
-p 10201:10201 \
corda/corda-zulu-5.0-snapshot:latest
```

## 18.5 Joining an existing Compatibility Zone

---

**Note:** Requirements: A Compatibility Zone, the Zone Trust Root and authorisation to join said Zone.

---

It is possible to use the image to automate the process of joining an existing Zone as detailed [here](#)

The first step is to obtain the Zone Trust Root, and place it within a directory. In the below example, the Trust Root is stored at /home/user/docker/certificates/network-root-truststore.jks. It is possible to configure the name of the Trust Root file by setting the TRUST\_STORE\_NAME environment variable in the container.

```
docker run -ti --net="host" \
-e MY_LEGAL_NAME="O=EXAMPLE,L=Berlin,C=DE" \
-e MY_PUBLIC_ADDRESS="corda.example-hoster.com" \
-e NETWORKMAP_URL="https://map.corda.example.com" \
-e DOORMAN_URL="https://doorman.corda.example.com" \
-e NETWORK_TRUST_PASSWORD="trustPass" \
-e MY_EMAIL_ADDRESS="cordauser@r3.com" \
-v /home/user/docker/config:/etc/corda \
-v /home/user/docker/certificates:/opt/corda/certificates \
corda/corda-zulu-5.0-snapshot:latest config-generator --generic
```

Several environment variables must also be passed to the container to allow it to register:

1. MY\_LEGAL\_NAME - The X500 to use when generating the config. This must be the same as registered with the Zone.
2. MY\_PUBLIC\_ADDRESS - The public address to advertise the node on.
3. NETWORKMAP\_URL - The address of the Zone's network map service (this should be provided to you by the Zone).
4. DOORMAN\_URL - The address of the Zone's doorman service (this should be provided to you by the Zone).
5. NETWORK\_TRUST\_PASSWORD - The password to the Zone Trust Root (this should be provided to you by the Zone).
6. MY\_EMAIL\_ADDRESS - The email address to use when generating the config. This must be the same as registered with the Zone.

There are some optional variables which allow customisation of the generated config:

1. MY\_P2P\_PORT - The port to advertise the node on (defaults to 10200). If changed, ensure the container is launched with the correct published ports.
2. MY\_RPC\_PORT - The port to open for RPC connections to the node (defaults to 10201). If changed, ensure the container is launched with the correct published ports.

Once the container has finished performing the initial registration, the node can be started as normal

```
docker run -ti \
    --memory=2048m \
    --cpus=2 \
    -v /home/user/docker/config:/etc/corda \
    -v /home/user/docker/certificates:/opt/corda/certificates \
    -v /home/user/docker/persistence:/opt/corda/persistence \
    -v /home/user/docker/logs:/opt/corda/logs \
    -v /home/user/corda/samples/bank-of-corda-demo/build/nodes/BankOfCorda/
    ↵cordapps:/opt/corda/cordapps \
    -p 10200:10200 \
    -p 10201:10201 \
    corda/corda-zulu-5.0-snapshot:latest
```

## 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](#).

### 19.1 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](#))

### 19.2 Deploying the Corda Network

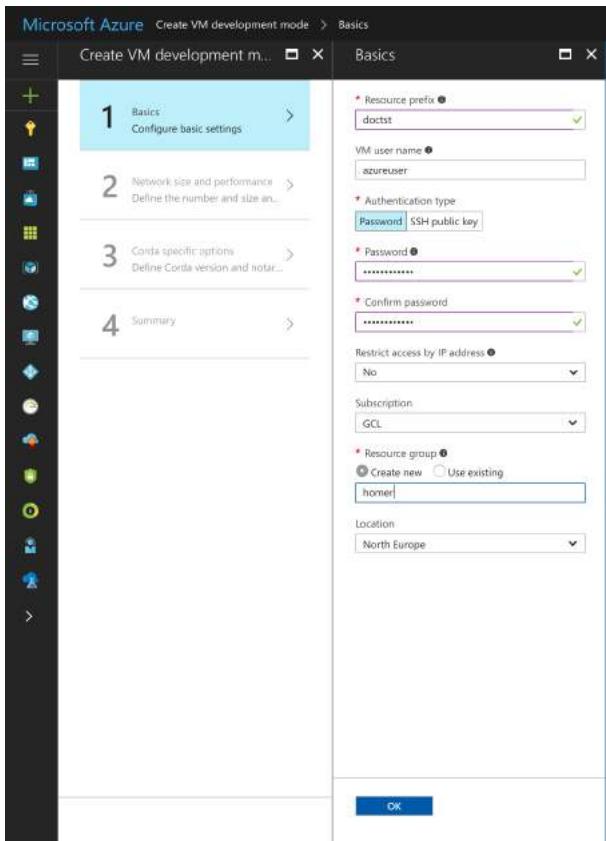
Browse to [portal.azure.com](#), login and search the Azure Marketplace for Corda and select ‘Corda Single Ledger Network’.

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 prerequisites, 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

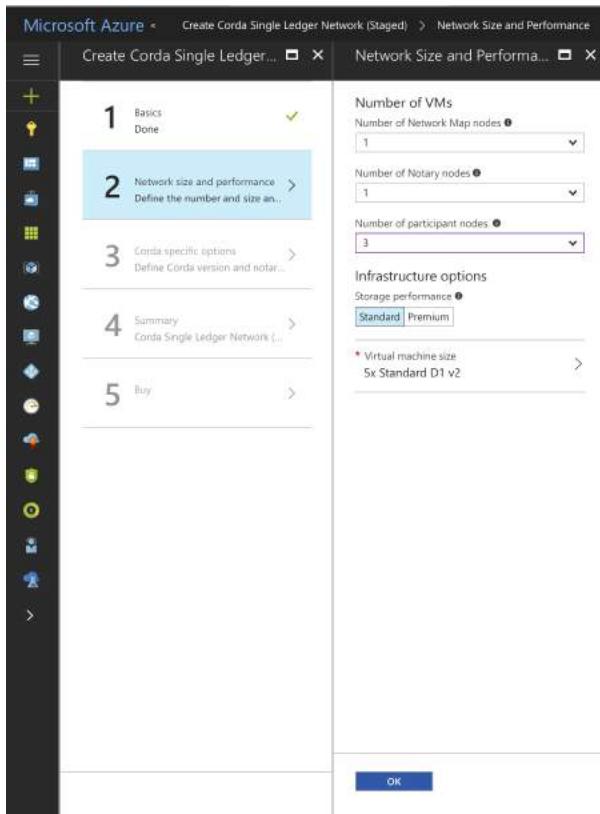


Click ‘OK’

#### 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

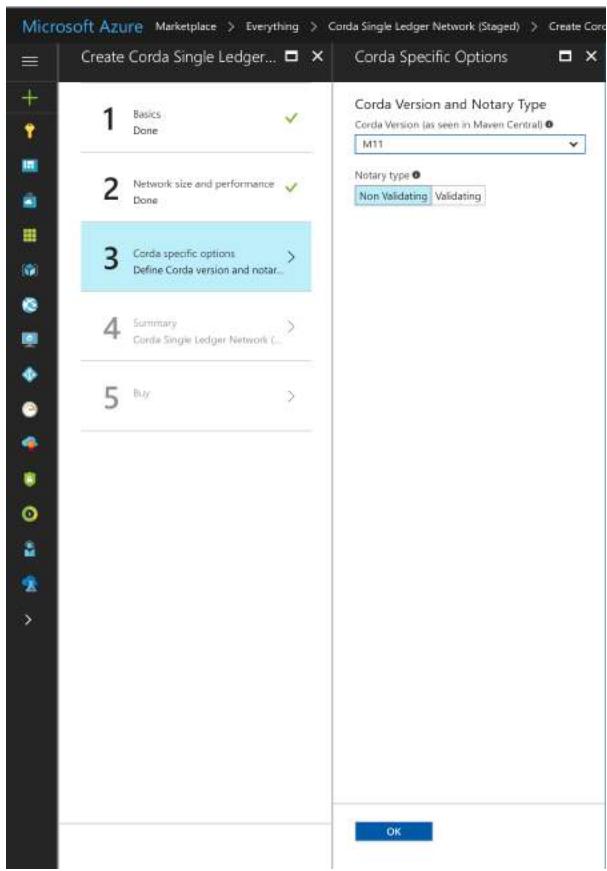


Click ‘OK’

### STEP 3: Corda Specific Options

Define the version of Corda you want on your nodes and the type of notary.

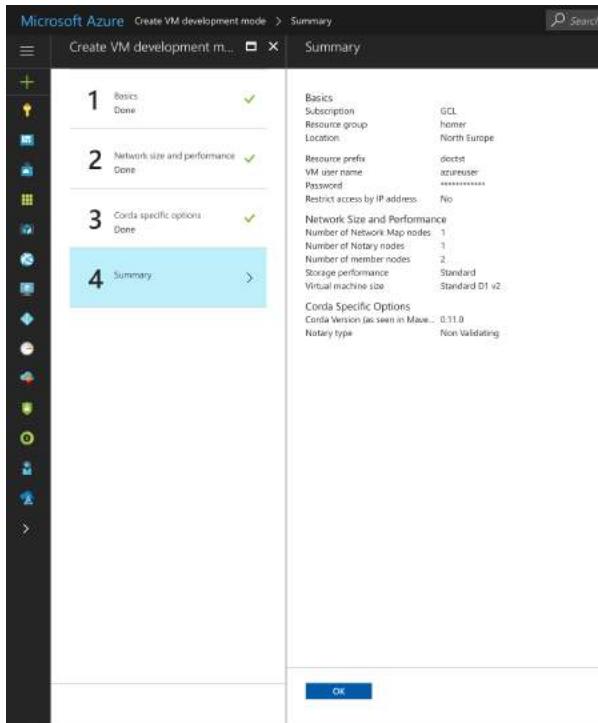
- **Corda version (as seen in Maven Central):** Select the version of Corda you want your nodes to use from the drop down list. The version numbers can be seen in [Maven Central](#), for example 0.11.0
- **Notary type:** Select either ‘Non Validating’ (notary only checks whether a state has been previously 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](#)



Click 'OK'

#### STEP 4: Summary

A summary of your selections is shown.



Click 'OK' for your selection to be validated. If everything is ok you will see the message 'Validation passed'

Click 'OK'

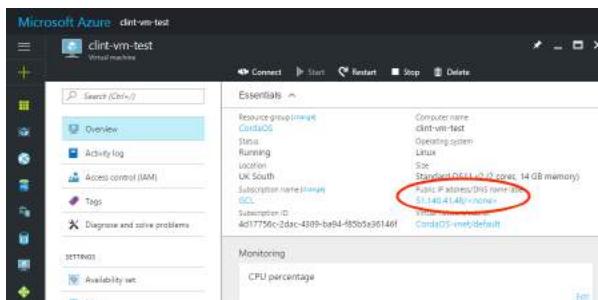
### STEP 5: Buy

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:



## 19.3 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:

Build the yo cordapp sample which you can find here: <https://github.com/corda/samples/blob/release-V4/yo-cordapp> and install it in the cordapp directory.

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.

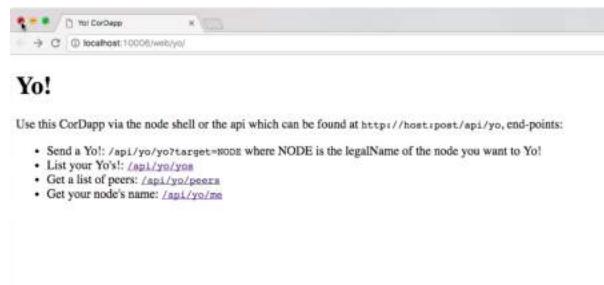
- **Verify the Yo! CorDapp is running**

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:



- **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_NetworkMap_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", "linearId" : { "externalId" : null, "id" : "a899aaef2-382b-4c8e-824e-0afdb190bd0" }, "contract" : { "legalContractReference" : "D118EC2116F70B65C948DC4812799B3AM317327FC8FB8A002D40864F437FB3EF" }, "participants" : [ "32hNaJauCq7qlacX4KaJvrzrNtiw8PsaueXKaJtMky4bd17nv6ypd74NZUrnq?" ], "notary" : "corda.notary.simple|Corda_0.10.1_Notary_Node_in_tstyo2", "escrowBranch" : null }, "ref" : { "txhash" : "A1D2F79F3BFC0C472BA6F3FBCEB80A3B635807F93AE7B115162663C16A559154C", "index" : 0 } } }
```

## 19.4 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.

## 19.5 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](#)

## 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](#).

### 20.1 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](#))

### 20.2 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.

### 20.3 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-xxxx-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](#).

## 20.4 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](#)

---

CHAPTER  
**TWENTYONE**

---

## 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.

### 21.1 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"]}
```

## 21.2 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.

## 21.3 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.

## 21.4 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
)

```

(continues on next page)

(continued from previous page)

```

)
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.

## 21.5 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.

---

CHAPTER  
TWENTYTWO

---

## SHELL EXTENSIONS FOR CLI APPLICATIONS

### 22.1 Installing shell extensions

Users of bash or zsh can install an alias and auto-completion for Corda applications that contain a command line interface. Run:

```
java -jar <name-of-JAR>.jar install-shell-extensions
```

Then, either restart your shell, or for bash users run:

```
. ~/ .bashrc
```

Or, for zsh run:

```
. ~/ .zshrc
```

You will now be able to run the command line application from anywhere by running the following:

```
<alias> --<option>
```

For example, for the Corda node, install the shell extensions using

```
java -jar corda-4.1.jar install-shell-extensions
```

And then run the node by running:

```
corda --<option>
```

### 22.2 Upgrading shell extensions

Once the shell extensions have been installed, you can upgrade them in one of two ways.

1. Overwrite the existing JAR with the newer version. The next time you run the application, it will automatically update the completion file. Either restart the shell or see [above](#) for instructions on making the changes take effect immediately.
2. If you wish to use a new JAR from a different directory, navigate to that directory and run:

```
java -jar <name-of-JAR>
```

Which will update the alias to point to the new location, and update command line completion functionality. Either restart the shell or see [above](#) for instructions on making the changes take effect immediately.

## 22.3 List of existing CLI applications

Description	Alias	JAR Name
<i>Corda node</i>	<code>corda --&lt;option&gt;</code>	<code>corda-4.1.jar</code>
<i>Network bootstrapper</i>	<code>bootstrapper --&lt;option&gt;</code>	<code>corda-tools-network-bootstrapper-4.1.jar</code>
<i>Standalone shell</i>	<code>corda-shell --&lt;option&gt;</code>	<code>corda-tools-shell-cli-4.1.jar</code>
<i>Blob inspector</i>	<code>blob-inspector --&lt;option&gt;</code>	<code>corda-tools-blob-inspector-4.1.jar</code>

---

CHAPTER  
**TWENTYTHREE**

---

## DETERMINISTIC CORDA MODULES

A Corda contract's verify function should always produce the same results for the same input data. To that end, Corda provides the following modules:

1. core-deterministic
2. serialization-deterministic
3. jdk8u-deterministic

These are reduced version of Corda's `core` and `serialization` modules and the OpenJDK 8 `rt.jar`, where the non-deterministic functionality has been removed. The intention here is that all CorDapp classes required for contract verification should be compiled against these modules to prevent them containing non-deterministic behaviour.

---

**Note:** These modules are only a development aid. They cannot guarantee determinism without also including deterministic versions of all their dependent libraries, e.g. `kotlin-stdlib`.

---

### 23.1 Generating the Deterministic Modules

**JDK 8 jdk8u-deterministic** is a “pseudo JDK” image that we can point the Java and Kotlin compilers to. It downloads the `rt.jar` containing a deterministic subset of the Java 8 APIs from the Artifactory.

To build a new version of this JAR and upload it to the Artifactory, see the `create-jdk8u` module. This is a standalone Gradle project within the Corda repository that will clone the `deterministic-jvm8` branch of Corda's [OpenJDK repository](#) and then build it. (This currently requires a C++ compiler, GNU Make and a UNIX-like development environment.)

**Corda Modules** `core-deterministic` and `serialization-deterministic` are generated from Corda's `core` and `serialization` modules respectively using both [ProGuard](#) and Corda's `JarFilter` Gradle plugin. Corda developers configure these tools by applying Corda's `@KeepForDJVM` and `@DeleteForDJVM` annotations to elements of `core` and `serialization` as described [here](#).

The build generates each of Corda's deterministic JARs in six steps:

1. Some *very few* classes in the original JAR must be replaced completely. This is typically because the original class uses something like `ThreadLocal`, which is not available in the deterministic Java APIs, and yet the class is still required by the deterministic JAR. We must keep such classes to a minimum!
2. The patched JAR is analysed by ProGuard for the first time using the following rule:

```
keep '@interface net.corda.core.KeepForDJVM { *; }'
```

ProGuard works by calculating how much code is reachable from given “entry points”, and in our case these entry points are the `@KeepForDJVM` classes. The unreachable classes are then discarded by ProGuard’s `shrink` option.

3. The remaining classes may still contain non-deterministic code. However, there is no way of writing a ProGuard rule explicitly to discard anything. Consider the following class:

```
@CordaSerializable
@KeepForDJVM
data class UniqueIdentifier @JvmOverloads @DeleteForDJVM constructor(
    val externalId: String? = null,
    val id: UUID = UUID.randomUUID()
) : Comparable<UniqueIdentifier> {
    ...
}
```

While CorDapps will definitely need to handle `UniqueIdentifier` objects, all of the secondary constructors generate a new random `UUID` and so are non-deterministic. Hence the next “determinising” step is to pass the classes to the `JarFilter` tool, which strips out all of the elements which have been annotated as `@DeleteForDJVM` and stubs out any functions annotated with `@StubOutForDJVM`. (Stub functions that return a value will throw `UnsupportedOperationException`, whereas `void` or `Unit` stubs will do nothing.)

4. After the `@DeleteForDJVM` elements have been filtered out, the classes are rescanned using ProGuard to remove any more code that has now become unreachable.
5. The remaining classes define our deterministic subset. However, the `@kotlin.Metadata` annotations on the compiled Kotlin classes still contain references to all of the functions and properties that ProGuard has deleted. Therefore we now use the `JarFilter` to delete these references, as otherwise the Kotlin compiler will pretend that the deleted functions and properties are still present.
6. Finally, we use ProGuard again to validate our JAR against the deterministic `rt.jar`:

```
task checkDeterminism(type: ProGuardTask, dependsOn: jdkTask) {
    injars metafix

    libraryjars deterministic_rt_jar

    configurations.deterministicLibraries.forEach {
        libraryjars it, filter: '!META-INF/versions/**'
    }

    keepattributes '*'
    dontpreverify
    dontobfuscate
    dontoptimize
    verbose

    keep 'class *'
}
```

This step will fail if ProGuard spots any Java API references that still cannot be satisfied by the deterministic `rt.jar`, and hence it will break the build.

## 23.2 Configuring IntelliJ with a Deterministic SDK

We would like to configure IntelliJ so that it will highlight uses of non-deterministic Java APIs as not found. Or, more specifically, we would like IntelliJ to use the `deterministic-rt.jar` as a “Module SDK” for deterministic modules rather than the `rt.jar` from the default project SDK, to make IntelliJ consistent with Gradle.

This is possible, but slightly tricky to configure because IntelliJ will not recognise an SDK containing only the `deterministic-rt.jar` as being valid. It also requires that IntelliJ delegate all build tasks to Gradle, and that Gradle be configured to use the Project’s SDK.

**Creating the Deterministic SDK** Gradle creates a suitable JDK image in the project’s `jdk8u-deterministic/jdk` directory, and you can configure IntelliJ to use this location for this SDK. However, you should also be aware that IntelliJ SDKs are available for *all* projects to use.

To create this JDK image, execute the following:

```
$ gradlew jdk8u-deterministic:copyJdk
```

Now select `File/Project Structure/Platform Settings/SDKs` and add a new JDK SDK with the `jdk8u-deterministic/jdk` directory as its home. Rename this SDK to something like “1.8 (Deterministic)”.

This *should* be sufficient for IntelliJ. However, if IntelliJ realises that this SDK does not contain a full JDK then you will need to configure the new SDK by hand:

1. Create a JDK Home directory with the following contents:

```
jre/lib/rt.jar
```

where `rt.jar` here is this renamed artifact:

```
<dependency>
  <groupId>net.corda</groupId>
  <artifactId>deterministic-rt</artifactId>
  <classifier>api</classifier>
</dependency>
```

2. While IntelliJ is *not* running, locate the `config/options/jdk.table.xml` file in IntelliJ’s configuration directory. Add an empty `<jdk>` section to this file:

```
<jdk version="2">
  <name value="1.8 (Deterministic)" />
  <type value="JavaSDK"/>
  <version value="java version "1.8.0""/>
  <homePath value=".. path to the deterministic JDK directory .."/>
  <roots>
  </roots>
</jdk>
```

3. Open IntelliJ and select `File/Project Structure/Platform Settings/SDKs`. The “1.8 (Deterministic)” SDK should now be present. Select it and then click on the `Classpath` tab. Press the “Add” / “Plus” button to add `rt.jar` to the SDK’s classpath. Then select the `Annotations` tab and include the same JAR(s) as the other SDKs.

### Configuring the Corda Project

1. Open the root `build.gradle` file and define this property:

```
buildscript {
    ext {
        ...
        deterministic_idea_sdk = '1.8 (Deterministic)'
        ...
    }
}
```

### Configuring IntelliJ

1. Go to File/Settings/Build, Execution, Deployment/Build Tools/Gradle, and configure Gradle's JVM to be the project's JVM.
2. Go to File/Settings/Build, Execution, Deployment/Build Tools/Gradle/Runner, and select these options:
  - Delegate IDE build/run action to Gradle
  - Run tests using the Gradle Test Runner
3. Delete all of the `out` directories that IntelliJ has previously generated for each module.
4. Go to View/Tool Windows/Gradle and click the Refresh all Gradle projects button.

These steps will enable IntelliJ's presentation compiler to use the deterministic `rt.jar` with the following modules:

- `core-deterministic`
- `serialization-deterministic`
- `core-deterministic:testing:common`

but still build everything using Gradle with the full JDK.

## 23.3 Testing the Deterministic Modules

The `core-deterministic:testing` module executes some basic JUnit tests for the `core-deterministic` and `serialization-deterministic` JARs. These tests are compiled against the deterministic `rt.jar`, although they are still executed using the full JDK.

The `testing` module also has two sub-modules:

**`core-deterministic:testing:data`** This module generates test data such as serialised transactions and elliptic curve key pairs using the full non-deterministic `core` library and JDK. This data is all written into a single JAR which the `testing` module adds to its classpath.

**`core-deterministic:testing:common`** This module provides the test classes which the `testing` and `data` modules need to share. It is therefore compiled against the deterministic API subset.

## 23.4 Applying `@KeepForDJVM` and `@DeleteForDJVM` annotations

Corda developers need to understand how to annotate classes in the `core` and `serialization` modules correctly in order to maintain the deterministic JARs.

---

**Note:** Every Kotlin class still has its own `.class` file, even when all of those classes share the same source file. Also, annotating the file:

```
@file:KeepForDJVM
package net.corda.core.internal
```

*does not* automatically annotate any class declared *within* this file. It merely annotates any accompanying Kotlin xxxKt class.

---

For more information about how JarFilter is processing the byte-code inside core and serialization, use Gradle's --info or --debug command-line options.

**Deterministic Classes** Classes that *must* be included in the deterministic JAR should be annotated as @KeepForDJVM.

```
@Target(FILE, CLASS)
@Retention(BINARY)
@CordaInternal
annotation class KeepForDJVM
```

To preserve any Kotlin functions, properties or type aliases that have been declared outside of a class, you should annotate the source file's package declaration instead:

```
@file:JvmName("InternalUtils")
@file:KeepForDJVM
package net.corda.core.internal

infix fun Temporal.until(endExclusive: Temporal): Duration = Duration.
    ↵between(this, endExclusive)
```

**Non-Deterministic Elements** Elements that *must* be deleted from classes in the deterministic JAR should be annotated as @DeleteForDJVM.

```
@Target(
    FILE,
    CLASS,
    CONSTRUCTOR,
    FUNCTION,
    PROPERTY_GETTER,
    PROPERTY_SETTER,
    PROPERTY,
    FIELD,
    TYPEALIAS
)
@Retention(BINARY)
@CordaInternal
annotation class DeleteForDJVM
```

You must also ensure that a deterministic class's primary constructor does not reference any classes that are not available in the deterministic rt.jar. The biggest risk here would be that JarFilter would delete the primary constructor and that the class could no longer be instantiated, although JarFilter will print a warning in this case. However, it is also likely that the "determinised" class would have a different serialisation signature than its non-deterministic version and so become unserialisable on the deterministic JVM.

Primary constructors that have non-deterministic default parameter values must still be annotated as @DeleteForDJVM because they cannot be refactored without breaking Corda's binary interface. The Kotlin compiler will automatically apply this @DeleteForDJVM annotation - along with any others - to all of the class's secondary constructors too. The JarFilter plugin can then remove the @DeleteForDJVM annotation from the primary constructor so that it can subsequently delete only the secondary constructors.

The annotations that `JarFilter` will “sanitise” from primary constructors in this way are listed in the plugin’s configuration block, e.g.

```
task jarFilter(type: JarFilterTask) {
    ...
    annotations {
        ...
        forSanitise = [
            "net.corda.core.DeleteForDJVM"
        ]
    }
}
```

Be aware that package-scoped Kotlin properties are all initialised within a common `<clinit>` block inside their host `.class` file. This means that when `JarFilter` deletes these properties, it cannot also remove their initialisation code. For example:

```
package net.corda.core

@DeleteForDJVM
val map: MutableMap<String, String> = ConcurrentHashMap()
```

In this case, `JarFilter` would delete the `map` property but the `<clinit>` block would still create an instance of `ConcurrentHashMap`. The solution here is to refactor the property into its own file and then annotate the file itself as `@DeleteForDJVM` instead.

**Non-Deterministic Function Stubs** Sometimes it is impossible to delete a function entirely. Or a function may have some non-deterministic code embedded inside it that cannot be removed. For these rare cases, there is the `@StubOutForDJVM` annotation:

```
@Target(
    CONSTRUCTOR,
    FUNCTION,
    PROPERTY_GETTER,
    PROPERTY_SETTER
)
@Retention(BINARY)
@CordaInternal
annotation class StubOutForDJVM
```

This annotation instructs `JarFilter` to replace the function’s body with either an empty body (for functions that return `void` or `Unit`) or one that throws `UnsupportedOperationException`. For example:

```
fun necessaryCode() {
    nonDeterministicOperations()
    otherOperations()
}

@StubOutForDJVM
private fun nonDeterministicOperations() {
    // etc
}
```

---

CHAPTER  
**TWENTYFOUR**

---

**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 apps to Corda 4](#).

## 24.1 Unreleased

- Fix a bug in Corda 4.0 that combined commands in `TransactionBuilder` if they only differed by the signers list. The behaviour is now consistent with prior Corda releases.
- Disabled the default loading of `hibernate-validator` as a plugin by hibernate when a CorDapp depends on it. This change will in turn fix the (<https://github.com/corda/corda/issues/4444>) issue, because nodes will no longer need to add `hibernate-validator` to the `\libs` folder. For nodes that already did that, it can be safely removed when the latest Corda is installed. One thing to keep in mind is that if any CorDapp relied on `hibernate-validator` to validate Querayable JPA Entities via annotations, that will no longer happen. That was a bad practice anyway, because the `ContractState` should be validated in the `Contract verify` method.

## 24.2 Version 4.0

- Fixed race condition between `NodeVaultService.trackBy` and `NodeVaultService.notifyAll`, where there could be states that were not reflected in the data feed returned from `trackBy` (either in the query's result snapshot or the observable).
- TimedFlows (only used by the notary client flow) will never give up trying to reach the notary, as this would leave the states in the notarisation request in an undefined state (unknown whether the spend has been notarised, i.e. has happened, or not). Also, retries have been disabled for single node notaries since in this case they offer no potential benefits, unlike for a notary cluster with several members who might have different availability.
- New configuration property `database.initialiseAppSchema` with values `UPDATE`, `VALIDATE` and `NONE`. The property controls the behavior of the Hibernate DDL generation. `UPDATE` performs an update of CorDapp schemas, while `VALIDATE` only verifies their integrity. The property does not affect the node-specific DDL handling and complements `database.initialiseSchema` to disable DDL handling altogether.
- `JacksonSupport.createInMemoryMapper` was incorrectly marked as deprecated and is no longer so.
- Standardised CorDapp version identifiers in jar manifests (aligned with associated cordapp Gradle plugin changes). Updated all samples to reflect new conventions.
- Introduction of unique CorDapp version identifiers in jar manifests for contract and flows/services CorDapps. Updated all sample CorDapps to reflect new conventions. See [CorDapp separation](#) for further information.

- Automatic Constraints propagation for hash-constrained states to signature-constrained states. This allows Corda 4 signed CorDapps using signature constraints to consume existing hash constrained states generated by unsigned CorDapps in previous versions of Corda.
- You can now load different CorDapps for different nodes in the node-driver and mock-network. This previously wasn't possible with the `DriverParameters.extraCordappPackagesToScan` and `MockNetwork.cordappPackages` parameters as all the nodes would get the same CorDapps. See `TestCordapp`, `NodeParameters.additionalCordapps` and `MockNodeParameters.additionalCordapps`.
- `DriverParameters.extraCordappPackagesToScan` and `MockNetwork.cordappPackages` have been deprecated as they do not support the new CorDapp versioning and MANIFEST metadata support that has been added. They create artificial CorDapp jars which do not preserve these settings and thus may produce incorrect results when testing. It is recommended `DriverParameters.cordappsForAllNodes` and `MockNetworkParameters.cordappsForAllNodes` be used instead.
- Fixed a problem with IRS demo not being able to simulate future dates as expected (<https://github.com/corda/corda/issues/3851>).
- Fixed a problem that was preventing `Cash.generateSpend` to be used more than once per transaction (<https://github.com/corda/corda/issues/4110>).
- Fixed a bug resulting in poor vault query performance and incorrect results when sorting.
- Improved exception thrown by `AttachmentsClassLoader` when an attachment cannot be used because its uploader is not trusted.
- Fixed deadlocks generated by starting flow from within `CordaServices`.
- Marked the `Attachment` interface as `@DoNotImplement` because it is not meant to be extended by CorDapp developers. If you have already done so, please get in contact on the usual communication channels.
- Added auto-acceptance of network parameters for network updates. This behaviour is available for a subset of the network parameters and is configurable via the node config. See [The network map](#) for more information.
- Deprecated `SerializationContext.withAttachmentsClassLoader`. This functionality has always been disabled by flags and there is no reason for a CorDapp developer to use it. It is just an internal implementation detail of Corda.
- Deprecated all means to directly create a `LedgerTransaction` instance, as client code is only meant to get hold of a `LedgerTransaction` via `WireTransaction.toLedgerTransaction`.
- Introduced new optional network bootstrapper command line options (`-register-package-owner`, `-unregister-package-owner`) to register/unregister a java package namespace with an associated owner in the network parameter `packageOwnership` whitelist.
- BFT-Smart and Raft notary implementations have been moved to the `net.corda.notary.experimental` package to emphasise their experimental nature. Note that it is not possible to preserve the state for both types of notaries when upgrading from V3 or an earlier Corda version.
- New “validate-configuration” sub-command to `corda.jar`, allowing to validate the actual node configuration without starting the node.
- CorDapps now have the ability to specify a minimum platform version in their `MANIFEST.MF` to prevent old nodes from loading them.
- CorDapps have the ability to specify a target platform version in their `MANIFEST.MF` as a means of indicating to the node the app was designed and tested on that version.
- Nodes will no longer automatically reject flow initiation requests for flows they don't know about. Instead the request will remain un-acknowledged in the message broker. This enables the recovery scenario whereby any

missing CorDapp can be installed and retried on node restart. As a consequence the initiating flow will be blocked until the receiving node has resolved the issue.

- `FinalityFlow` is now an inlined flow and requires `FlowSession`s to each party intended to receive the transaction. This is to fix the security problem with the old API that required every node to accept any transaction it received without any checks. Existing CorDapp binaries relying on this old behaviour will continue to function as previously. However, it is strongly recommended CorDapps switch to this new API. See [Upgrading apps to Corda 4](#) for further details.
- For similar reasons, `SwapIdentitiesFlow`, from `confidential-identities`, is also now an inlined flow. The old API has been preserved but it is strongly recommended CorDapps switch to this new API. See [Upgrading apps to Corda 4](#) for further details.
- Introduced new optional network bootstrapper command line option (`-minimum-platform-version`) to set as a network parameter
- Vault storage of contract state constraints metadata and associated vault query functions to retrieve and sort by constraint type.
- New overload for `CordaRPCClient.start()` method allowing to specify target legal identity to use for RPC call.
- Case insensitive vault queries can be specified via a boolean on applicable SQL criteria builder operators. By default queries will be case sensitive.
- Getter added to `CordaRPCOps` for the node's network parameters.
- The RPC client library now checks at startup whether the server is of the client libraries major version or higher. Therefore to connect to a Corda 4 node you must use version 4 or lower of the library. This behaviour can be overridden by specifying a lower number in the `CordaRPCClientConfiguration` class.
- Removed experimental feature `CordformDefinition`
- Added new overload of `StartedMockNode.registerInitiatedFlow` which allows registering custom initiating-responder flow pairs, which can be useful for testing error cases.
- “app”, “rpc”, “p2p” and “unknown” are no longer allowed as uploader values when importing attachments. These are used internally in security sensitive code.
- Change type of the `checkpoint_value` column. Please check the upgrade-notes on how to update your database.
- Removed buggy `:serverNameTablePrefix:` configuration.
- `freeLocalHostAndPort`, `freePort`, and `getFreeLocalPorts` from `TestUtils` have been deprecated as they don't provide any guarantee the returned port will be available which can result in flaky tests. Use `PortAllocation.Incremental` instead.
- Docs for `IdentityService.assertOwnership` updated to correctly state that an `UnknownAnonymousPartyException` is thrown rather than `IllegalStateException`.
- The Corda JPA entities no longer implement `java.io.Serializable`, as this was causing persistence errors in obscure cases. Java serialization is disabled globally in the node, but in the unlikely event you were relying on these types being Java serializable please contact us.
- Remove all references to the out-of-process transaction verification.
- The class `carpenter` has a “lenient” mode where it will, during deserialisation, happily synthesis classes that implement interfaces that will have unimplemented methods. This is useful, for example, for object viewers. This can be turned on with `SerializationContext.withLenientCarpenter`.
- Added a `FlowMonitor` to log information about flows that have been waiting for IO more than a configurable threshold.

- H2 database changes: \* The node's H2 database now listens on `localhost` by default. \* The database server address must also be enabled in the node configuration. \* A new `h2Settings` configuration block supersedes the `h2Port` option.
- Improved documentation PDF quality. Building the documentation now requires `LaTeX` to be installed on the OS.
- Add `devModeOptions.allowCompatibilityZone` to re-enable the use of a compatibility zone and `devMode`
- Fixed an issue where `trackBy` was returning `ContractStates` from a transaction that were not being tracked. The unrelated `ContractStates` will now be filtered out from the returned `Vault.Update`.
- Introducing the flow hospital - a component of the node that manages flows that have errored and whether they should be retried from their previous checkpoints or have their errors propagate. Currently it will respond to any error that occurs during the resolution of a received transaction as part of `FinalityFlow`. In such a scenario the receiving flow will be parked and retried on node restart. This is to allow the node operator to rectify the situation as otherwise the node will have an incomplete view of the ledger.
- Fixed an issue preventing out of process nodes started by the `Driver` from logging to file.
- Fixed an issue with `CashException` not being able to deserialize after the introduction of AMQP for RPC.
- Removed `-Xmx` VM argument from Explorer's Capsule setup. This helps avoiding out of memory errors.
- New `killFlow` RPC for killing stuck flows.
- Shell now kills an ongoing flow when `CTRL+C` is pressed in the terminal.
- Add check at startup that all persisted Checkpoints are compatible with the current version of the code.
- `ServiceHub` and `CordaRPCOps` can now safely be used from multiple threads without incurring in database transaction problems.
- Doorman and NetworkMap url's 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 certificates](#) for details.
- Improved audit trail for `FinalityFlow` and related sub-flows.
- Notary client flow retry logic was improved to handle validating flows better. Instead of re-sending flow messages the entire flow is now restarted after a timeout. The relevant node configuration section was renamed from `p2pMessagingRetry`, to `flowTimeout` to reflect the behaviour change.
- The node's configuration is only printed on startup if `devMode` is `true`, avoiding the risk of printing passwords in a production setup.
- `NodeStartup` will now only print node's configuration if `devMode` is `true`, avoiding the risk of printing passwords in a production setup.
- SLF4J's MDC will now only be printed to the console if not empty. No more log lines ending with “{}”.
- `WireTransaction.Companion.createComponentGroups` has been marked as `@CordaInternal`. It was never intended to be public and was already internal for Kotlin code.
- RPC server will now mask internal errors to RPC clients if not in `devMode`. `Throwable``'s` implementing `ClientRelevantError` will continue to be propagated to clients.
- RPC Framework moved from Kryo to the Corda AMQP implementation [Corda-847]. This completes the removal of Kryo from general use within Corda, remaining only for use in flow checkpointing.
- Set `co.paralleluniverse.fibers.verifyInstrumentation=true` in `devMode`.
- Node will now gracefully fail to start if one of the required ports is already in use.

- Node will now gracefully fail to start if `devMode` is true and `compatibilityZoneURL` is specified.
- Added smart detection logic for the development mode setting and an option to override it from the command line.
- Changes to the JSON/YAML serialisation format from `JacksonSupport`, which also applies to the node shell:
  - `WireTransaction` now nicely outputs into its components: `id`, `notary`, `inputs`, `attachments`, `outputs`, `commands`, `timeWindow` and `privacySalt`. This can be deserialized back.
  - `SignedTransaction` is serialised into `wire` (i.e. currently only `WireTransaction` tested) and `signatures`, and can be deserialized back.
- 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.
- Due to ongoing work the experimental interfaces for defining custom notary services have been moved to the internal package. CorDapps implementing custom notary services will need to be updated, see `samples/notary-demo` for an example. Further changes may be required in the future.
- Configuration file changes:
  - Added program line argument `on-unknown-config-keys` to allow specifying behaviour on unknown node configuration property keys. Values are: [FAIL, IGNORE], default to FAIL if unspecified.
  - Introduced a placeholder for custom properties within `node.conf`; the property key is “custom”.
  - The deprecated web server now has its own `web-server.conf` file, separate from `node.conf`.
  - Property keys with double quotes (e.g. “key”) in `node.conf` are no longer allowed, for rationale refer to [Node configuration](#).
  - The `issuableCurrencies` property is no longer valid for `node.conf`. Instead, it has been moved to the finance workflows CorDapp configuration.
- Added public support for creating `CordaRPCClient` using SSL. For this to work the node needs to provide client applications a certificate to be added to a truststore. See [Using the client RPC API](#)
- The node RPC broker opens 2 endpoints that are configured with `address` and `adminAddress`. RPC Clients would connect to the address, while the node will connect to the adminAddress. Previously if `ssl` was enabled for RPC the `adminAddress` was equal to `address`.
- Upgraded H2 to v1.4.197
- Shell (embedded available only in dev mode or via SSH) connects to the node via RPC instead of using the `CordaRPCOps` object directly. To enable RPC connectivity ensure node’s `rpcSettings.address` and `rpcSettings.adminAddress` settings are present.
- Changes to the network bootstrapper:
  - The `whitelist.txt` file is no longer needed. The existing `network parameters` file is used to update the current contracts whitelist.
  - The CorDapp jars are also copied to each nodes’ `cordapps` directory.
- Errors thrown by a Corda node will now be reported to a calling RPC client with attention to serialization and obfuscation of internal data.
- Serializing an inner class (non-static nested class in Java, inner class in Kotlin) will be rejected explicitly by the serialization framework. Prior to this change it didn’t work, but the error thrown was opaque (complaining about too few arguments to a constructor). Whilst this was possible in the older Kryo implementation (Kryo passing

null as the synthesised reference to the outer class) as per the Java documentation [here](#) we are disallowing this as the paradigm in general makes little sense for contract states.

- Node can be shut down abruptly by `shutdown` function in `CordaRPCOps` or gracefully (draining flows first) through `gracefulShutdown` command from shell.
- API change: `net.corda.core.schemas.PersistentStateRef` fields (`index` and `txId`) are now non-nullable. The fields were always effectively non-nullable - values were set from non-nullable fields of other objects. The class is used as database Primary Key columns of other entities and databases already impose those columns as non-nullable (even if JPA annotation `nullable=false` was absent). In case your Cordapps use this entity class to persist data in own custom tables as non Primary Key columns refer to [Upgrading apps to Corda 4](#) for upgrade instructions.
- Adding a public method to check if a public key satisfies Corda recommended algorithm specs, `Crypto.validatePublicKey(java.security.PublicKey)`. For instance, this method will check if an ECC key lies on a valid curve or if an RSA key is  $\geq 2048$  bits. This might be required for extra key validation checks, e.g., for Doorman to check that a CSR key meets the minimum security requirements.
- Table name with a typo changed from `NODE_ATTACHMENTS_CONTRACTS` to `NODE_ATTACHMENT_CONTRACTS`.
- Node logs a warning for any `MappedSchema` containing a JPA entity referencing another JPA entity from a different `MappedSchema`. The log entry starts with “Cross-reference between `MappedSchemas`”. API: Persistence documentation no longer suggests mapping between different schemas.
- Upgraded Artemis to v2.6.2.
- Introduced the concept of “reference input states”. A reference input state is a `ContractState` which can be referred to in a transaction by the contracts of input and output states but whose contract is not executed as part of the transaction verification process and is not consumed when the transaction is committed to the ledger but is checked for “current-ness”. In other words, the contract logic isn’t run for the referencing transaction only. It’s still a normal state when it occurs in an input or output position. *This feature is only available on Corda networks running with a minimum platform version of 4.*
- A new wrapper class over `StateRef` is introduced, called `ReferenceStateRef`. Although “reference input states” are stored as `StateRef` objects in `WireTransaction`, we needed a way to distinguish between “input states” and “reference input states” when required to filter by object type. Thus, when one wants to filter-in all “reference input states” in a `FilteredTransaction` then he/she should check if it is of type `ReferenceStateRef`.
- Removed type parameter `U` from `tryLockFungibleStatesForSpending` to allow the function to be used with `FungibleState` as well as `FungibleAsset`. This `_might_` cause a compile failure in some obscure cases due to the removal of the type parameter from the method. If your CorDapp does specify types explicitly when using this method then updating the types will allow your app to compile successfully. However, those using type inference (e.g. using Kotlin) should not experience any changes. Old CorDapp JARs will still work regardless.
- `issuer_ref` column in `FungibleStateSchema` was updated to be nullable to support the introduction of the `FungibleState` interface. The `vault_fungible_states` table can hold both `FungibleAssets` and `FungibleStates`.
- CorDapps built by `corda-gradle-plugins` are now signed and sealed JAR files. Signing can be configured or disabled, and it defaults to using the Corda development certificate.
- Finance CorDapps are now built as sealed and signed JAR files. Custom classes can no longer be placed in the packages defined in either finance Cordapp or access its non-public members.
- Finance CorDapp was split into two separate apps: `corda-finance-contracts` and `corda-finance-workflows`. There is no longer a single cordapp which provides both. You need to have both JARs installed in the node simultaneously for the app to work however.

- All sample CorDapps were split into separate apps: workflows and contracts to reflect new convention. It is recommended to structure your CorDapps this way, see [Upgrading apps to Corda 4](#) on upgrading your CorDapp.
- The format of the shell commands' output can now be customized via the node shell, using the `output-format` command.
- The `node_transaction_mapping` database table has been folded into the `node_transactions` database table as an additional column.
- Logging for P2P and RPC has been separated, to make it easier to enable all P2P or RPC logging without hand-picking loggers for individual classes.
- Vault Query Criteria have been enhanced to allow filtering by state relevancy. Queries can request all states, just relevant ones, or just non relevant ones. The default is to return all states, to maintain backwards compatibility. Note that this means apps running on nodes using Observer node functionality should update their queries to request only relevant states if they are only expecting to see states in which they participate.
- Postgres dependency was updated to version 42.2.5
- Test `CordaService`s can be installed on mock nodes using `UnstartedMockNode.installCordaService`.
- The finance-contracts demo CorDapp has been slimmed down to contain only that which is relevant for contract verification. Everything else has been moved to the finance-workflows CorDapp:
  - The cash selection logic. `AbstractCashSelection` is now in `net.corda.finance.contracts.asset` so any custom implementations must now be defined in `META-INF/services/net.corda.finance.workflows.asset.selection.AbstractCashSelection`.
  - The jackson annotations on `Expression` have been removed. You will need to use `FinanceJSONSupport.registerFinanceJSONMappers` if you wish to preserve the JSON format for this class.
  - The various utility methods defined in `Cash` for creating cash transactions have been moved to `net.corda.finance.workflows.asset.CashUtils`. Similarly with `CommercialPaperUtils` and `ObligationUtils`.
  - Various other utilities such as `GetBalances` and the test calendar data.

The only exception to this is `Interpolator` and related classes. These are now in the [IRS demo workflows CorDapp](#).

- Vault states are migrated when moving from V3 to V4: the relevancy column is correctly filled, and the state party table is populated. Note: This means Corda can be slow to start up for the first time after upgrading from V3 to V4.

End of changelog.