r3.

corda

Corda Developer Certification Exam Study Guide

R3's Developer Relations • August 2022

Index

What is Corda? 4

Privacy 4

Development 4

Regulator Compliance 4

Scalability and Sustainability 4

Settlement 4

Security 4

States 5

Contracts 6

Commands 7

Flows 7

Transactions 8

Consensus 9

Network 10

Framework 11

Nodes 12

Reference states 13

Attachments 13

Oracles 14

Timestamps 14

More info 14

IOU CorDapp 15

WhistleBlower CorDapp 15

Observable States CorDapp 16

HeartBeat CorDapp 16

Negotiation CorDapp 16

Blacklist CorDapp 16

Recommended resources to assist with certification. 17

The Corda certification exam is designed for test-takers to demonstrate their technical knowledge in developing distributed Corda applications (CorDapps). The questions seek to test an individual's knowledge on key concepts and features of the Corda platform, and ability to address practical challenges associated with building a CorDapp.

What is Corda?

Corda's development framework enables the building of future-proof apps quickly in financial services and other regulated markets.

Privacy

Corda shares data only between the counterparties of a transaction. Even the communication protocol itself is invisible to other members of the network.

Development

Smart contracts are legally bound and written in any JVM compatible language, taking advantage of the vast ecosystem of tooling and libraries.

Regulator Compliance

Corda capabilities are grounded in legal constructs and compatible with existing and emerging standards and regulations.

Scalability and Sustainability

P2P architecture enables high levels of network scalability and throughput. Transactions don't need to be sequential, increasing the overall efficiency of the system. Approximate energy or joules consumed per transaction: 24.6.

Settlement

Consensus model guarantees that assets have deterministic settlement finality because it is based on validation and uniqueness.

Security

In order to participate in a CorDapp, each entity must be granted access to do so and tied to a legal entity.

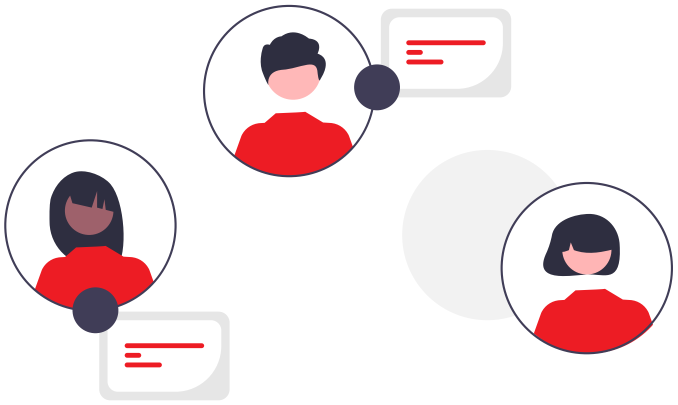
States

A state in Corda represents a fact known by one or more parties. States are not meant to be shared. State classes may request flows to be started at given times allowing for event scheduling.

Imagine a class which implements OwnableState interface. partyA will store the state after it gets committed to the ledger:

```
class MyCommand : CommandData, TypeOnlyCommandData()
data class MyState(val partyA : AbstractParty, val partyB : AbstractParty) : OwnableState
override val owner = partyA
override val participants = listOf(partyA, partyB)
override fun withNewOwner(newOwner : AbstractParty) = CommandAndState(MyCommand(), copy()
```

All FungibleAsset are issued by a single party, and these things can't be split and merged.



Fields of any ContractState are not stored in separate database columns when persisted. For this reason, it is impossible to run any kind of SQL-like queries against these states. The ContractState interface defines the participants field.

When creating a state pointer for any OwnableState, it is required to use the LinearPointer class because an Ownable state should only have one owner. The LinearState interface defines the linearId field. Fungible asset, owned by a single party, which can be split and merged with other states of the same type should be represented using OwnableState.

A QueryableState interface must be implemented so that the state is stored in a human-readable format in the vault. The vault is a database where each node store their states. Tables are required in a database to store vault information.
If you want the field to possibly contain an anonymous identity, you define a field in a state as having the type AbstractParty instead of the type party.

Contracts

A contract in Corda represent the sets of constraints that govern the evolution of state objects. Corda contracts code are included in transactions' attachments in Corda.

Contracts must be evaluated in a deterministic sandbox to protect against non-determinism in the evaluation of whether the transaction is valid or not.

The Signature Accept Contract Constraint is the default contract constraints that `TransactionBuilder` uses if the CorDapp was signed when built. The `Contract.verify()` method returns: `Void/Unit`.

A a node-administrator cease a contract upgrade by running the `ContractUpgradeFlow.Deauthorise` flow. With `ContractUpgradeFlow()`, it is possible to transmute a Cat state into a Dog state, provided that all participants in the Cat state agree to the change.

A transaction uniquely identify the contracts to use to verify a transaction by hash of the JAR file containing the contract and the fully qualified contract class name.

A Corda contract verifies:

- Each transaction state specifies a contract type
- The contract of every input state considers it to be valid
- The contract of every output state considers it to be valid

To achieve a Corda contract upgrade you have to accomplish the following:

- ✓ Pre-authorize multiple implementations of the contract ahead of time, using constraints.
- ✓ Create a special contract upgrade transaction and getting all participants of a state to sign it using the contract upgrade flows.

Keep in mind that It is NOT possible to allow non-deterministic Java libraries, such as `DateTime`, to be used in contracts if to fulfill business logic.

Counterparty node operator should always 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.

Commands

A command includes an indicator on the intent of the transaction and a list of public keys required to sign the transaction. Commands are require in Corda transactions and always has one or more signer(s).

The required signers of a transaction are decided based on the signers listed on the commands.

The required signers are added to the command with the `addCommand()` method inside a flow class where transaction is built. Command constructors inside the contract class where the different command classes are created.

You can't open multiple sessions with the same party within the same flow context.

The `sendAndReceive()` method won't trigger flow suspension until a response is received.

The party which initiates a flow is most likely responsible for creating the output States that are the proposed changes to the ledger.

Flows

Flows can invoke sub-flows, and a library of flows is provided to automate common tasks Flows can provide a progress tracker that indicates which step they are up to. Flows always return a result and could be paused the framework guarantee eventual finality.

To preserve flows across a node reboot each flow is automatically serialized and persisted to disk to be eventually finished. Remember that flows remain in a suspended state indefinitely.

`TransactionBuilder` should be used in the `Flow` Corda class.

Anonymous-to-well-known party mappings are resolved as a part of `IdentitySyncFlow`. `InitiatedBy` is an annotation used to specify which initiator flow a response flow responds to.

The `FinalityFlow` is a pre-defined flow used to notarise a transaction and record it in every participant/owner's vault.

A state's notary changed once the state has been issued. Any node can initiate a state's notary using the `NotaryChangeFlow`.

An exception in a flow within a `FlowException` instance needs to be wrapped when you want the exception to be propagated back to the node that is suspended and waiting for your flow to respond because of a sent or received call.

If your node receives a message from a flow for which it has not registered a response flow, the node ignores the message.

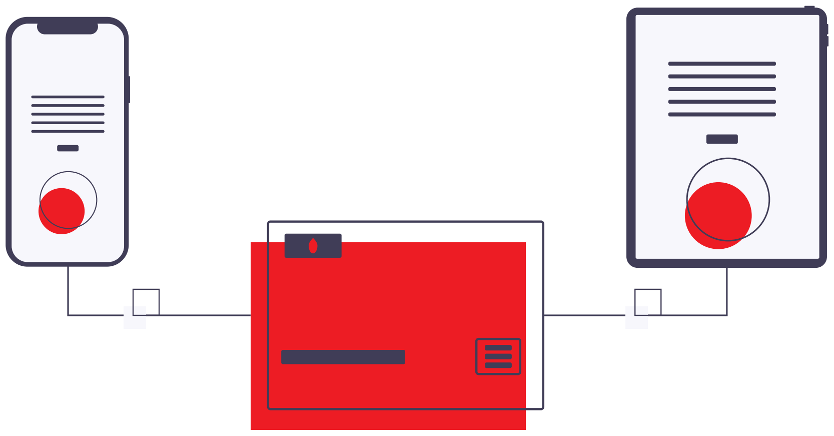
The StartableByRPC annotation is needed to be triggered from an RPC call.

Transactions

A transaction in Corda represents a proposal for a number of participants to update their ledgers. Any party on the network can propose a transaction in Corda. The number of participants in a transaction is unlimited.

A transaction is filtered to provide advanced privacy by hiding certain components of the transaction from users who do not need to see them. A transaction can be broadcasted to the Corda network.

A transaction have to be resolved to a LedgerTransaction before being verified because the inputs must be converted from state references into state objects and the attachments must be converted from hashes into actual attachments. A transaction can have only one notary and one combined input and output states. Thus, a transaction creates zero or more output states.



A transaction become immutable after it has one or more digital signatures, and becomes a SignedTransaction. The attachment's hash is included in the transaction. The hash of a transaction is the root hash of the transaction's Merkle tree.
When instantiating a TransactionBuilder the reference to a Notary is an optional argument. A

transaction is verifiable within a TransactionBuilder, after partial signing, and after complete signing.

Access to the private key is restricted to the node's internals; thus, you can't get access to your node's private key to sign a transaction.

Each state is linked to a contract using a TransactionState.

The hash of the transaction that created the state and the state's index in the outputs of that transaction is included in a StateRef. All state changes must be valid. Any invalid state transition will result in a failed transaction.

To build a FilteredTransaction, a WireTransaction and filtering rules are needed.

A transaction's signatures can't be checked for validity from within a contract's verify method because the signatures are not included in the transaction. Contract.verify() is the only code that runs when verifying a transaction.

Consensus

There must be consensus that a proposed transaction is valid before you can add it to the ledger. Blockchains use consensus mechanisms to achieve agreement, trust, and security across decentralized networks.

When the transaction has no input states and no timewindow a notary is not required to notarize a transaction.

Each node receives the evidence of all of the transaction history during verification. When signing a transaction, the input state references, the TimeWindow, and the transaction's notary are seen by a non-validating notary. Checking that the signatures on the transaction are valid is a type of verification that must go in the flow.

Each notary maintains a list of state references that have already been consumed to ensure each output state is only consumed once.

The original notary have to sign when reassigning states with the NotaryChangeFlow.

Transactions can't have inputs states that are assigned to different notaries.

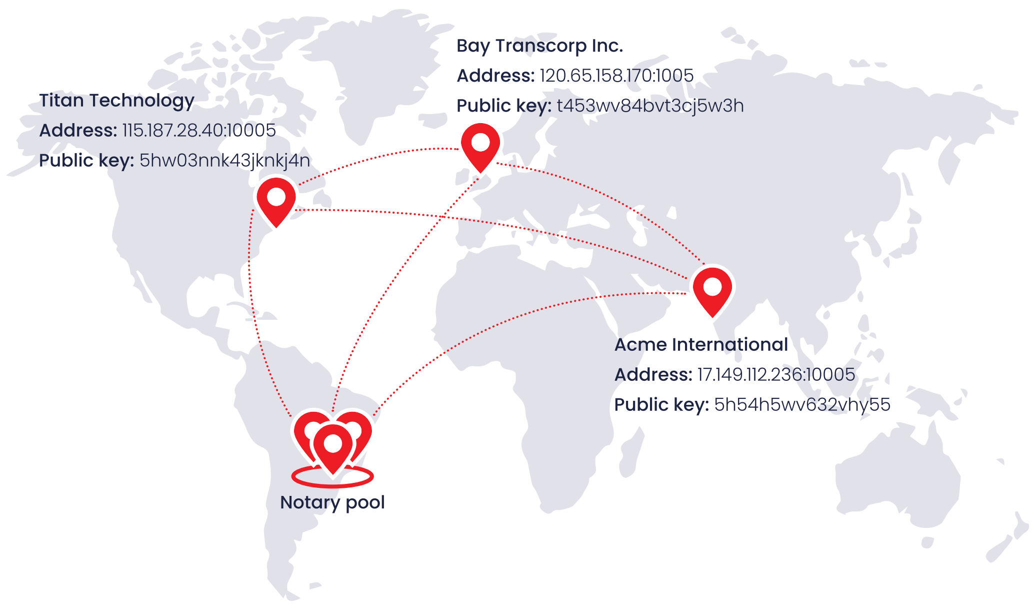
The entire historical chain doesn't have to be traced backward and re-verified in order to validate a transaction. Corda doesn't have strict consensus policies that inhibits any variation with the consensus algorithms.

The different notary clusters that are in the same network doesn't have to run the same consensus algorithm. Many notary clusters can co-exist on the same network.

Every state has an appointed notary cluster. A transaction will only be notarized if its input states are equivalent to the output states of the previous transaction.

Network

Corda Network is a publicly-available Internet of Corda nodes operated by network participants. Each node is identified by a certificate issued by the network's identity service, and is also discoverable on a network map.


Bay Transcorp Inc.
Address: 120.65.158.170:1005
Public key: t453wv84bvt3cj5w3h
Titan Technology
Address: 115.187.28.40:10005
Public key: 5hw03nnk43jknkj4n
Acme International
Address: 17.149.112.236:10005
Public key: 5h54h5wv632vhy55
Notary pool

The number of adjacent nodes that outside the current network is not included inside the network map for the nodes on the network. Information is shared in Corda on a peer-to-peer (P2P) basis and few participants can join a Corda network.

The private key of a "Confidential identity" is never revealed to any other parties in the network. X509 is the certificate with which a node is able to verify the identity of the owner of a public key.

A node needs three Java-style Keystores to successfully run in a bootstrap network. A `sslkeystore.jks` stores the node's TLS key pair and certificate.

AMQP is the Corda's P2P messaging protocol. If a node goes offline, messages will be queue and then retried until the remote node has acknowledged the message.

The regulator node is not required to run a Corda network and the doorman controls admission to a Corda network.

Framework

Serialization converts objects into a stream of bytes. Deserialization, the reverse process, creates objects from a stream of bytes. These two processes take place every time nodes pass objects to each other as messages, when the node sends objects to or from RPC clients, and when you store transactions in the database.

Classes can be serialized by whitelisting them in Corda by implementing the `SerialisationWhitelist` interface. Third party classes have to be explicitly added to the serialization whitelist in order to be sent over the network.

In classic Java serialization, any class on the JVM classpath can be deserialized. This can be exploited by adding a stream of malicious bytes to the large set of third-party libraries that are added to the classpath as part of a JVM application's dependencies. Corda strictly controls which classes can be deserialized (and, proactively, serialized) by insisting that each (de)serializable class is on a whitelist of allowed classes.

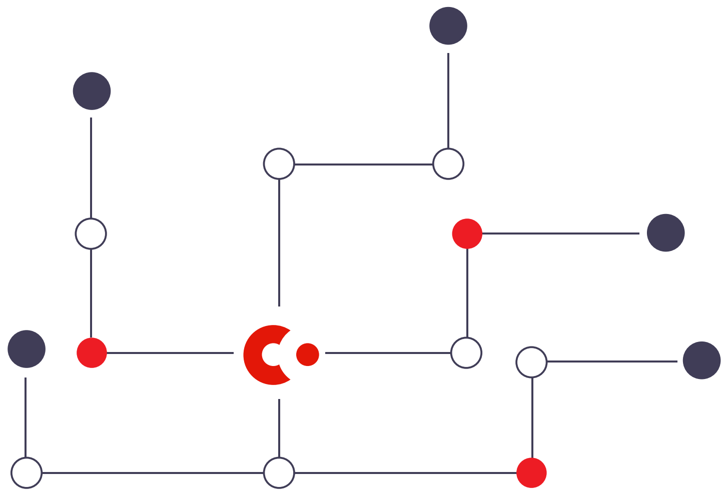
A node can have multiple CorDapps running at the same time.

TLS is used to encrypt communications between nodes. CordaSerializable marks a class as being eligible to be sent and received between nodes as part of a flow.

SwapIdentitiesFlow exchanges the Confidential Identity between different parties.

Nodes

Since all nodes will run on a single machine, all ports need to be unique and assigned. If you did not assign ports, the nodes would all use the default of 10002 and that would create a port conflict on a single machine. The P2P address is not mentioned so it will default to localhost. Now, if you wanted to run your services on different machines, you would update the configuration with externally resolvable names or IPs under the label p2pAddress:, like so p2pAddress "192.168.0.15:10006".


Corda Nodes run in a JVM

A CorDapp is installed on a node by placing it in the node's CorDapps folder. 2GB & 1GB is the suggested minimum host RAM and JVM Heap size for hosting a node. Data does need to be backed up: each participant must back up their own data.

Note that a continent cannot be used for node identity.

In Corda, user-facing client and external integration into the node is done via CordaRPCClient. When a node sees a transaction, only the visible states are stored in its vault. Each node store their states in the vault.

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.

Reference states

Reference states that are used in transactions are stored in the vault. When reference states are called in a transaction, the reference states' contract need to be executed as part of the transaction verification process.

Reference states can be referred to in a transaction by the contracts of input and output states.

Each reference state is checked for "current-ness" by the specified notary cluster for the containing transaction.

`StateAndRef.referenced()` should be called in order to obtain `ReferencedStateAndRef`.

Possible use-cases of reference state includes:

- Set FX rate as reference states so different transactions can be validated according to the most current data.
- Financial instrument reference data provided by the issuer (Corda node), so other parties can refer to issuance origin for metadata.
- A distributed KYC application that the KYC data can be distributed as reference data states, so other parties can build their transaction be conditional on this reference data.

Attachments

Attachments are intended to be reused across transactions. They are stored in JAR files and presented as hashes of the original files inside a transaction.

When a node sees an attachment it resolves it by retrieving the attachment from its own storage or requests it from the counterparty.

The receiver node downloads the physical files via RPC client calls.

Attachment metadata can be queried in a similar way to the vault.

Oracles

Oracles provides data external to the ledger into a transaction and signs the transaction. Some examples of data are, the price of a stock at a point of time, an agreed-upon interest rate at a point in time, and weather conditions at a particular place. Oracle services can be used to propose and verify transactions.

Timestamps

A contract ensures that a transaction is only valid if it is fully signed before time X by adding a TimeWindow to the transaction and getting the notary to sign the transaction within that TimeWindow.
Time may be specified for the following ranges:

1 **Before** 2 **After** 3 **Between**

More info

The following items are based on general Corda concepts and keyphrases.

- The default transaction size is stored in the network parameter file.
- The Java Integrated Development Environment (IDE) named IntelliJ will assist you in navigating the code with simple clicks and hints.
- Corda transactions may include attachments that are included with the transaction but are not part of the transaction itself. These are .ZIP or .JAR files attached to the transactions.

- Pkcs12 and Jks are the key storetypes that Corda accept.
- If a bilateral flow is kicked off between two parties, first checkpointed by Party A and then by Party B, do those flows share an ID? E.g. - if I search through the checkpoints on both nodes, will I be able to match them?
  No, the ID of the checkpoint is based on the ID of the `FlowStateMachine`. This state machine is not shared between the two nodes.
- `gracefulShutdown` is the first command that you will run when upgrading your node from the older version to the latest version of Corda.
- Corda is designed to help enterprises achieve consensus about shared facts. The facts are generally related to contracts that closely model legal contracts, and transactions that explain the evolving states of the contracts.

IOU CorDapp

The following items are related to the IOU CorDapp in Java:

```
// Sign the transaction.
final SignedTransaction partSignedTx = getServiceHub().signInitialTransaction(txBuilder);
```

The code snippet above shows how the txBuilder object is used to complete the initial transaction signing. Whoever is initiating the transaction, “me” in this case is making the initial signing of the transaction.

`ContractState` is the return type of: `output = stx.getTx().getOutputs().get(0).getData();`

The testing code works by starting a mock network and running the unit tests to assess the different parts of the application.

WhistleBlower CorDapp

The following items are related to the WhistleBlower CorDapp:

```
val confidentialIdentities =
subFlow(SwapIdentitiesFlow( counterpartySession,
GENERATE_CONFIDENTIAL_IDS.childProgressTracker()))
```

The code snippet above returns a `LinkedHashMap` of `<Party, AnonymousParty>` pair.

Observable States CorDapp

The following items are related to the Observable States CorDapp:

To achieve the observable feature in the Observable States sample, the state of the observer was distributed using `FinalityFlow` and the state to the observer was manually distributed using the helper flow.

The `recordTransaction()` method handles the writing to the ledger when manually distributing states to the observer using helper flow.

HeartBeat CorDapp

The following items are related to the HeartBeat CorDapp Sample:

In this CorDapp sample, there is a notary running because there is a transaction going on between parties.

The `HeartState` uses the `SchedulableState` class.

Negotiation CorDapp

The following items are related to the Negotiation Sample:

The Linear ID of the output state that was generated from the transaction is the return value when you initiate a proposal using `ProposalFlow`.
The same Linear ID that was used in the input state is the ID attached in the output of the transaction when the counterparty runs `ModificationFlow`.

In `AcceptanceFlow`, the output of the transaction is a `TradeState` that can only be generated during the `AcceptanceFlow`. The only restriction that is coded in the `AcceptanceFlow`'s responder flow is that only the proposee can accept a proposal.

A single `TradeState` state will get recorded into the Corda vault when a business that requires running the following flows once and in order: `ProposalFlow` > `ModificationFlow` > `AcceptanceFlow`.

Blacklist CorDapp

The following items are related to the Blacklist CorDapp.

- `verify()` in the `AgreementContract` `.kt` handles the exclusion of blacklisted entities.
- The `checkTransaction()` method checks if the received transaction has an output `AgreementState` in the `AgreeFlow`.

Recommended resources to assist with certification

Corda training: training.corda.net

Documentation: developer.r3.com/docs/

Blogs: developer.r3.com/blog/category/corda

Videos: developer.r3.com/videos

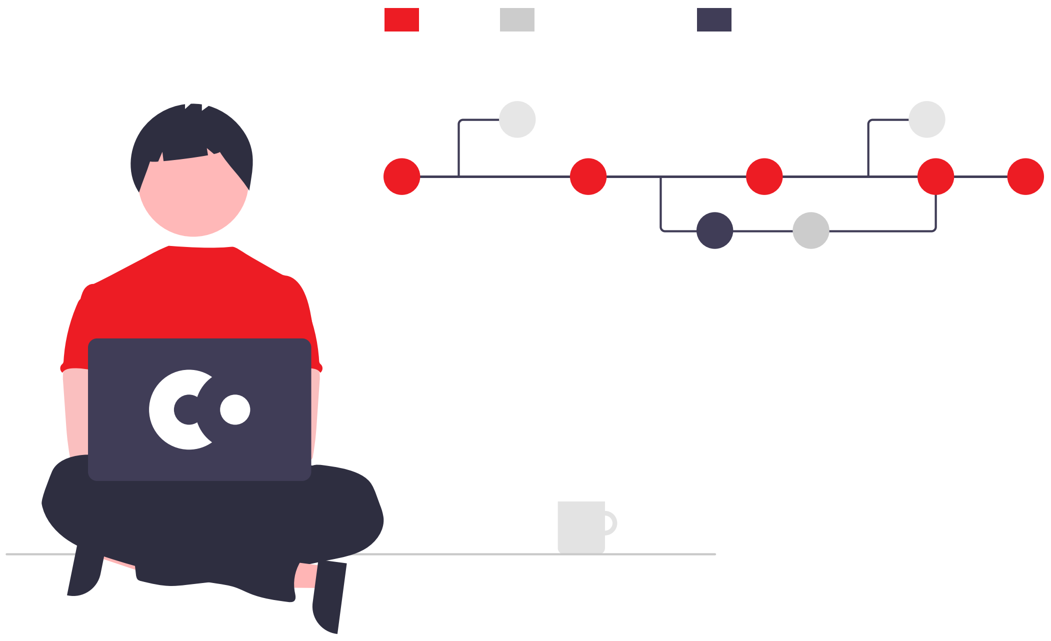
Community: community.r3.com

Open-source code repository: github.com/corda/corda

Open-source CorDapp sample repository: github.com/corda/samples

Public Slack: slack.corda.net

Good luck and happy coding!



R3 is a leading provider of enterprise technology and services that enable direct, digital collaboration in regulated industries where trust is critical. Multi-party solutions developed on our platforms harness the "Power of 3"—R3's trust technology, connected networks and regulated markets expertise—to drive market innovation and improve processes in banking, capital markets, global trade and insurance.

As one of the first companies to deliver both a private, distributed ledger technology (DLT) application platform and confidential computing technology, R3 empowers institutions to realize the full potential of direct digital collaboration. We maintain one of the largest DLT production ecosystems in the world connecting over 400 institutions, including global systems integrators, cloud providers, technology firms, software vendors, corporates, regulators, and financial institutions from the public and private sectors.

For more information, visit
www.r3.com and developer.r3.com

r3.com • developer.r3.com