**Get Started with Corda**

Blockchain has come a long way. The idea of distributed ledger technology has solved many pain points that the current organizations go through. The main aim is to become more efficient, secure, and scalable without investing millions of dollars.

Corda is an open source blockchain platform that provides a global blockchain ecosystem. It offers strict privacy with the use of smart contracts. It also helps the business to transact directly and provide a cost-effective approach to streamline business operations.

The main difference is how Corda handles the transactions. It is a DLT that wants to improve blockchain feasibility, speed, and scalability by not having a complete ledger with each peer. If a transaction can be completed between two peers without informing the whole network, then Corda will use that approach. It is also designed to include other parties that make it easy to follow the transaction and see its validity. It is a distributed ledger technology for the business.

**Tools**

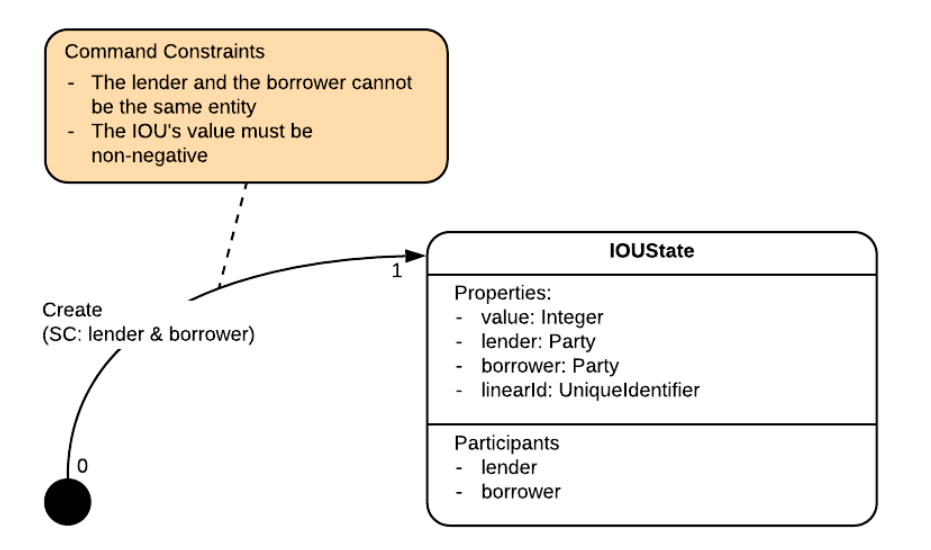
Before we get started with the Corda tutorial, we will need the following tools. There are also blockchain tools that developers really love to use frequently.

* Gradle
* Git
* IntelliJ
* Command-line
* Oracle JDK
* Java Version -> Java8

As of the integrated development environment, we are going to use the IntelliJ. It will simplify our development.

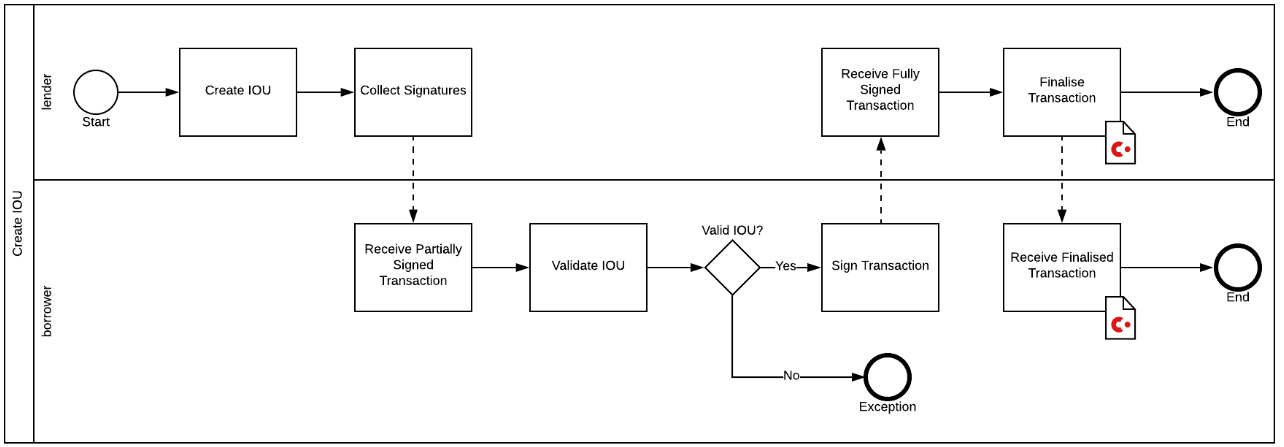
**The Corda Objects**

* **Nodes –** A node is a piece of software that participates in a Corda system or network and runs Cordapps.
* **States –** The aggregate of all states held by all nodes of the network is the distributed ledger state. There is no central ledger, and not all nodes know all states, so the overall ledger is subjective from the perspective of each participant.
* **Transactions -** Transactions are what consume states and produce new states. They are atomic. Transactions either complete entirely or have no effect. There are not partially complete, “in-flight” transactions although they do have a lifecycle with various stages.
* **Contracts –** A transaction is contractually valid if all its input and output states are acceptable according to the contract. Contracts are written in Java or Kotlin. Contract execution is deterministic, and transaction acceptance is based on the transaction’s contents alone.

****

* **Flows –** 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.

To achieve this consensus, the flow goes through several steps:



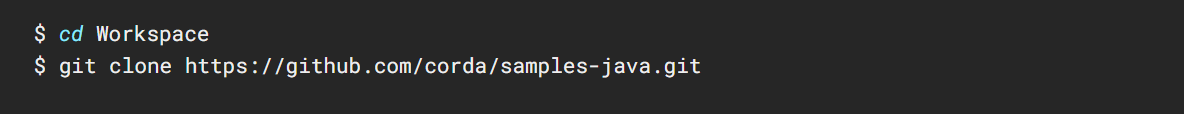
* **Consensus -** Consensus is the process by which all parties achieve certainty about the shared states. Corda applies two types of consensuses:
* **Validity Consensus** – It requires contractual validity of the transaction and all its dependencies. This is checked by each required signer before they sign the transaction.
* **Uniqueness Consensus** – It prevents double-spends. This is only checked by a notary service.
* **The Parties** – Let us take a quick look at the parties.

Party: Both lender and borrower are of type Party. If you look at Party’s definition, it starts to get a bit cryptic, with its default constructor.

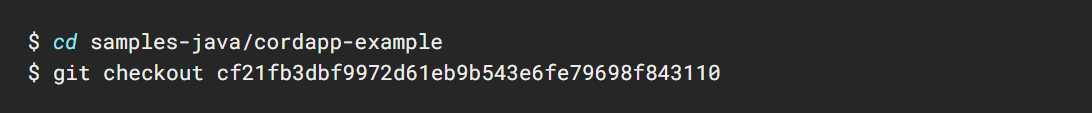
**Steps to Navigate the Cordapp-Project**

1. Cloning Git repository

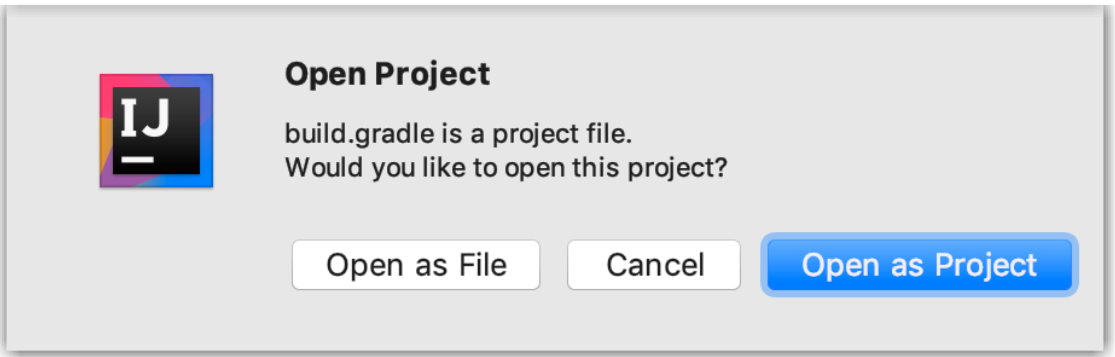
The best way to start our journey is to run an example of CorDapp in our development environment. The good news is that Corda offers an example CorDapp that we can download and run locally.



This samples repository contains a lot of example projects; however, we will focus only on cordapp-example. To ensure our version matches, we need to add an extra step:



1. The easiest way to open the project with IntelliJ is:
2. Open IntelliJ.
3. Click on the menu File->Open.
4. Navigate to the Cordapp-Example.
5. Select the root build. gradle project file.
6. Click Open.
7. A popup appears, now in the popup click Open as Project.



1. Click on Import Gradle project and let it finish, and that’s it.

**Run the Cordapp Project**

Running the project in a test and development context is a 3-step process.

1. Declare the configuration.
2. Create the nodes configuration files.
3. Launch the executables.

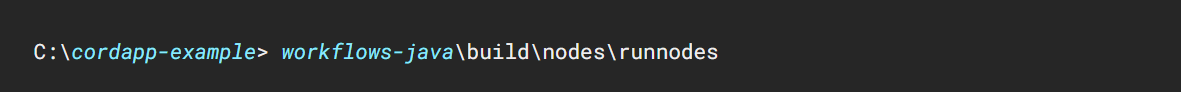
**Build the configuration files**

Be sure that we are in cordapp-example working directory. Let us run this build task with:

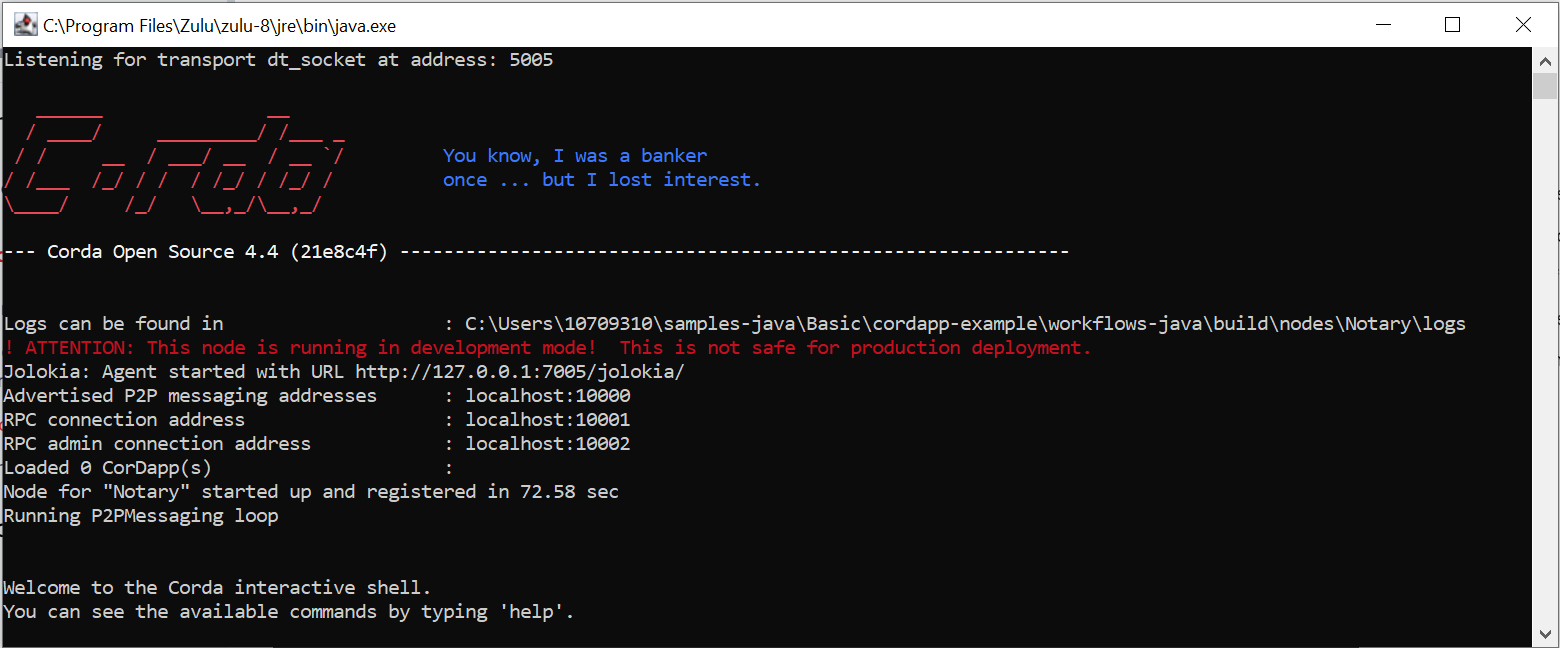


**Run the Nodes**

Now that our nodes have been deployed, now it’s time to run the following command:



We should get a 4 pop-up windows; do not move focus away from them until all 4 show the welcome message:

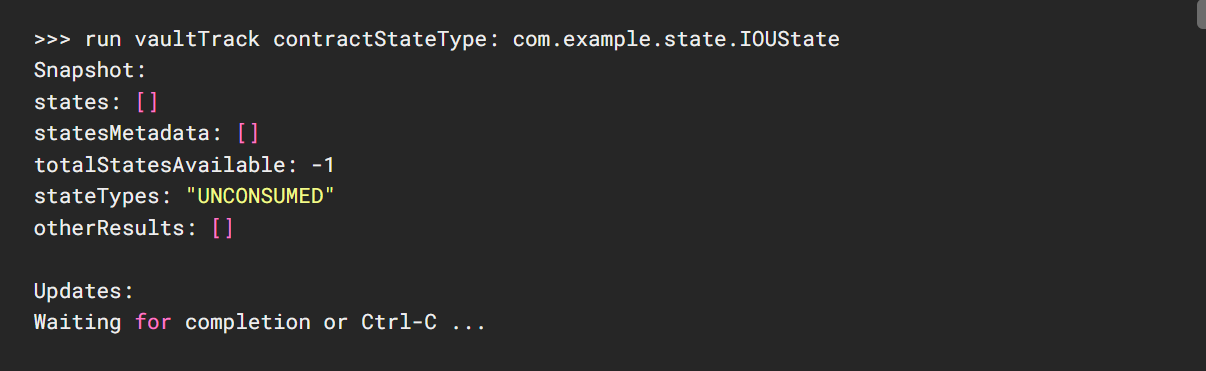


**Interact with the Nodes**

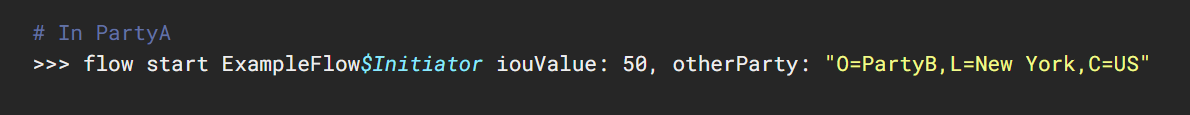
We are going to create our first IOU where PartyB owes 50 to PartyA. Because of how the flows are coded:

1. You need to initiate the flow from the lender, i.e., PartyA.
2. PartyB will receive it.
3. PartyC will be oblivious.

To show points 2 and 3, let’s prepare PartyB and PartyC with an IOUState tracker. On the interactive shells of both nodes of PartyB and PartyC, run:



Then head to the interactive shell of PartyA and launch the flow



It takes a few seconds because the processes are started lazily. Notice the last line, that looks like:



Head over to PartyC and see that the tracker has not seen anything, but on PartyB we should see something like:



Nice, it worked as expected:

* Nothing was consumed: Observation 1: consumed: [].
* A single state was created, or produced, with the expected values. This concludes our first interaction with a flow from the command line.

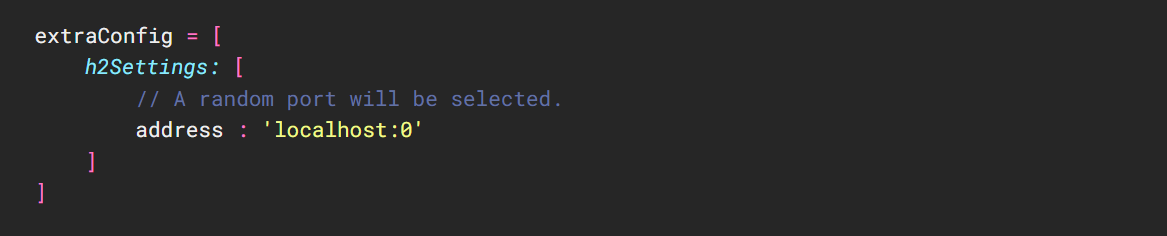
**Look into the database (vault)**

The database used in a development network is H2. To access it, we need to:

* Install some H2 software.
* Reconfigure our nodes.

**Configure the nodes**

Stop our running nodes by typing bye into each shell. Then, inside workflow- java/build.gradle’s deployNodes.nodeDefaults, set the H2 database port by adding the below section:



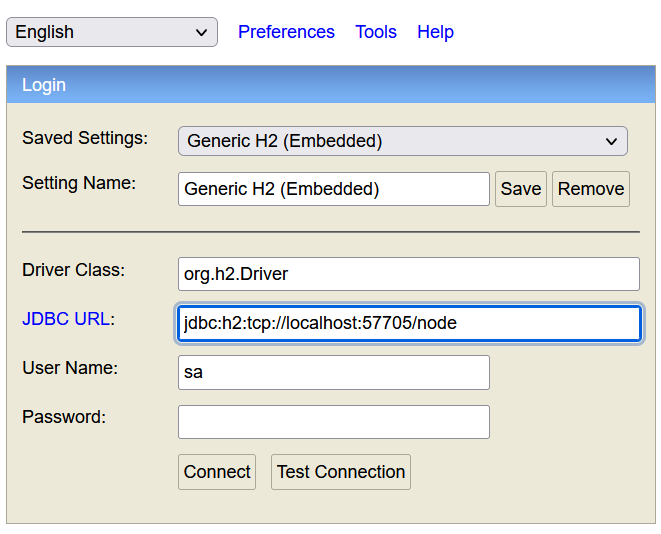
Then:

1. Build the nodes as we did earlier.
2. Run the nodes as we did earlier.

On each running node shells, we should see a new line, like:



Now start the H2 console, run sh /bin/h2/bin/h2.sh



**Empty states**

Start by connecting to our notary on our first H2 tab. Click on **NODE-NOTARY\_REQUEST\_LOG**:

This should populate the query field with **SELECT \* FROM NODE\_NOTARY\_REQUEST\_LOG**. Press Run. There are no rows. This means that as of now you have not had any state that was consumed.

On another tab, connect to PartyA. Click on **VAULT\_STATES** and Run. Again, empty.

**Create Our third IOU**

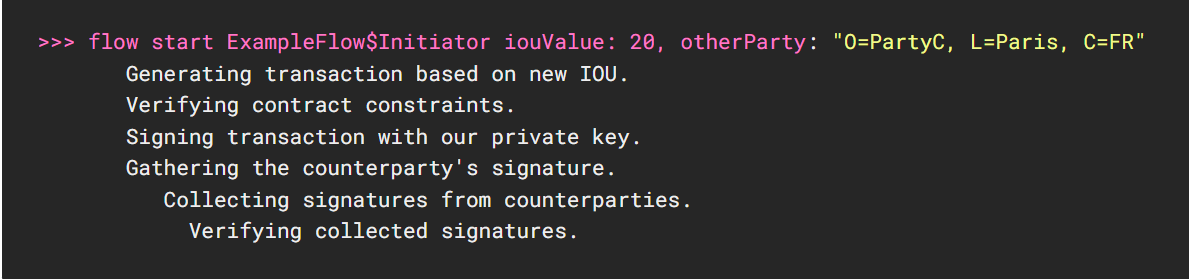
* Again, we will use PartyA as the lender, but PartyC will now be the borrower. This time, we are going to experiment with checkpointed state machine. We will remember that a checkpoint happens when a flow is waiting for something coming from elsewhere. In the IOU case,

PartyA has to wait for PartyC to return the transaction signed. It is at this point that PartyA will checkpoint. PartyA will checkout whether PartyC is running or stopped.

* However, for PartyA’s checkpointed state to lost long enough for you to see it, you will first stop PartyC. Type bye into PartyC’s shell. While you are at it, also stop the Notary.



Now back in PartyA’s shell:



It remains stuck at **Collecting signatures from counterparties.** That makes sense because PartyC is down. You will note that the steps are made explicit in the code itself.

Do not interrupt PartyA’s flow. Return to your H2 console connected to PartyA and run the following queries:

* SELECT \* FROM IOU\_STATES
* SELECT \* FROM NODE\_CHECKPOINTS

Now, start PartyC again. We will start the individual node from the **workflows-java/build/nodes** folder that was created by. **/gradle deployNodes**

