Suporte ao desenvolvimento

Curso Blockchain Developer - Turma JUN2018

04 de Julho de 2018

Material produzido por bbchain.

Projeto 1 - Aula 2 - corDapp IOU

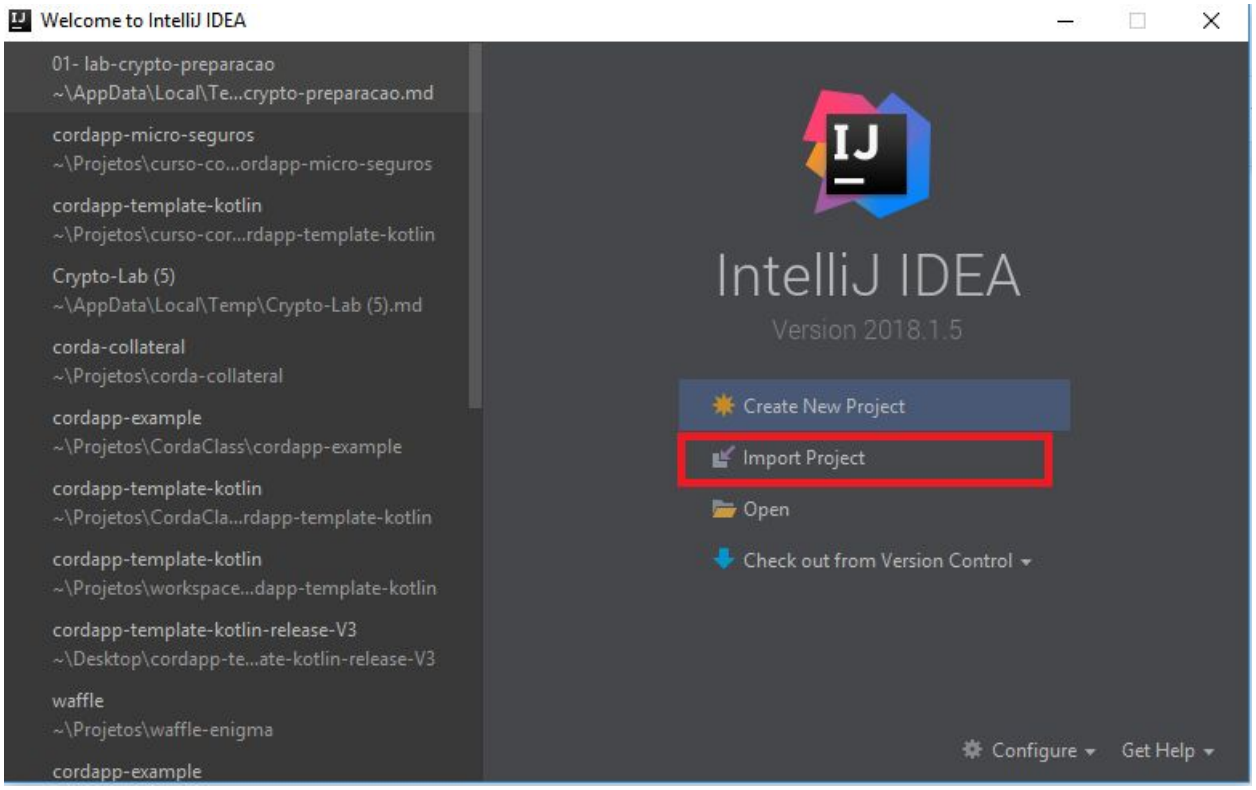
Agora que já conhecemos a estrutura de deploy do Corda, vamos dar uma olhada no código do corDapp IOU.

Nesta aula vamos ver:

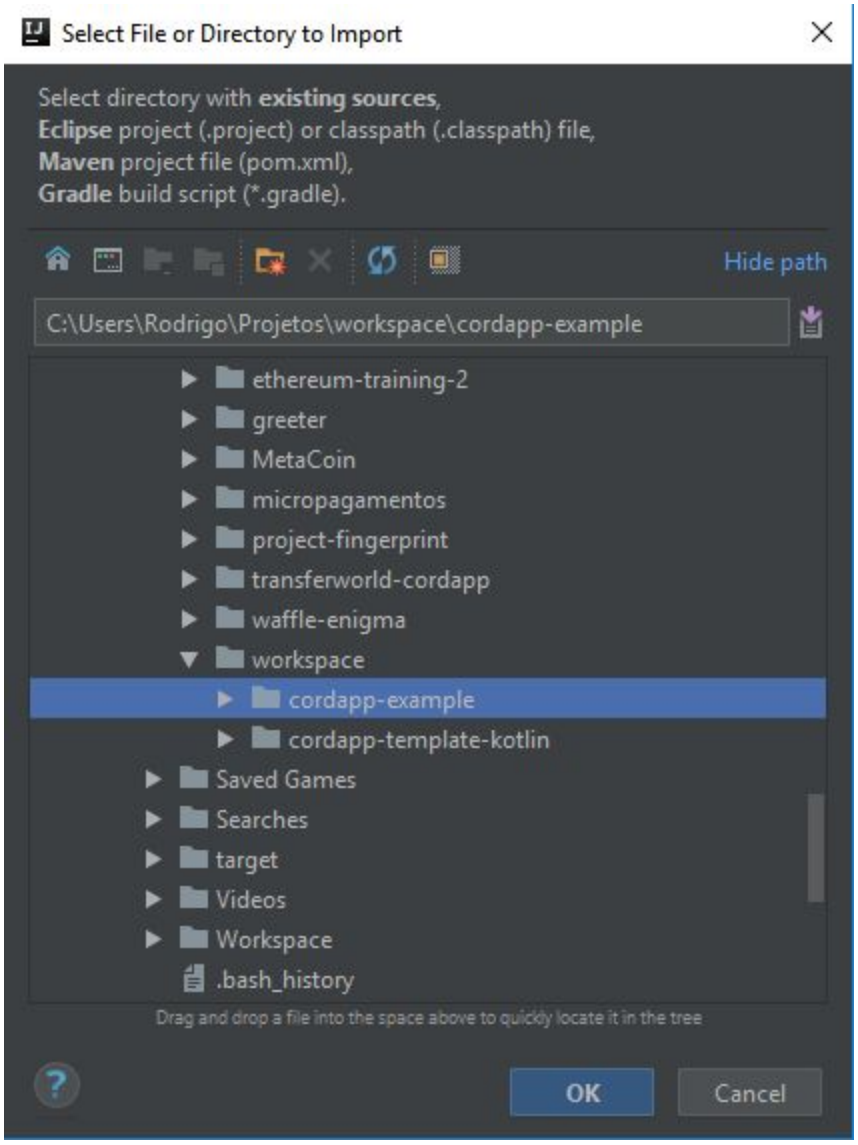
- Conceitos:
  - State
  - Contract
  - Flow
- RPC em Corda
- APIs em Corda
- Teste em Corda

Antes de iniciarmos, vamos abrir o cordapp-example no Intellij.

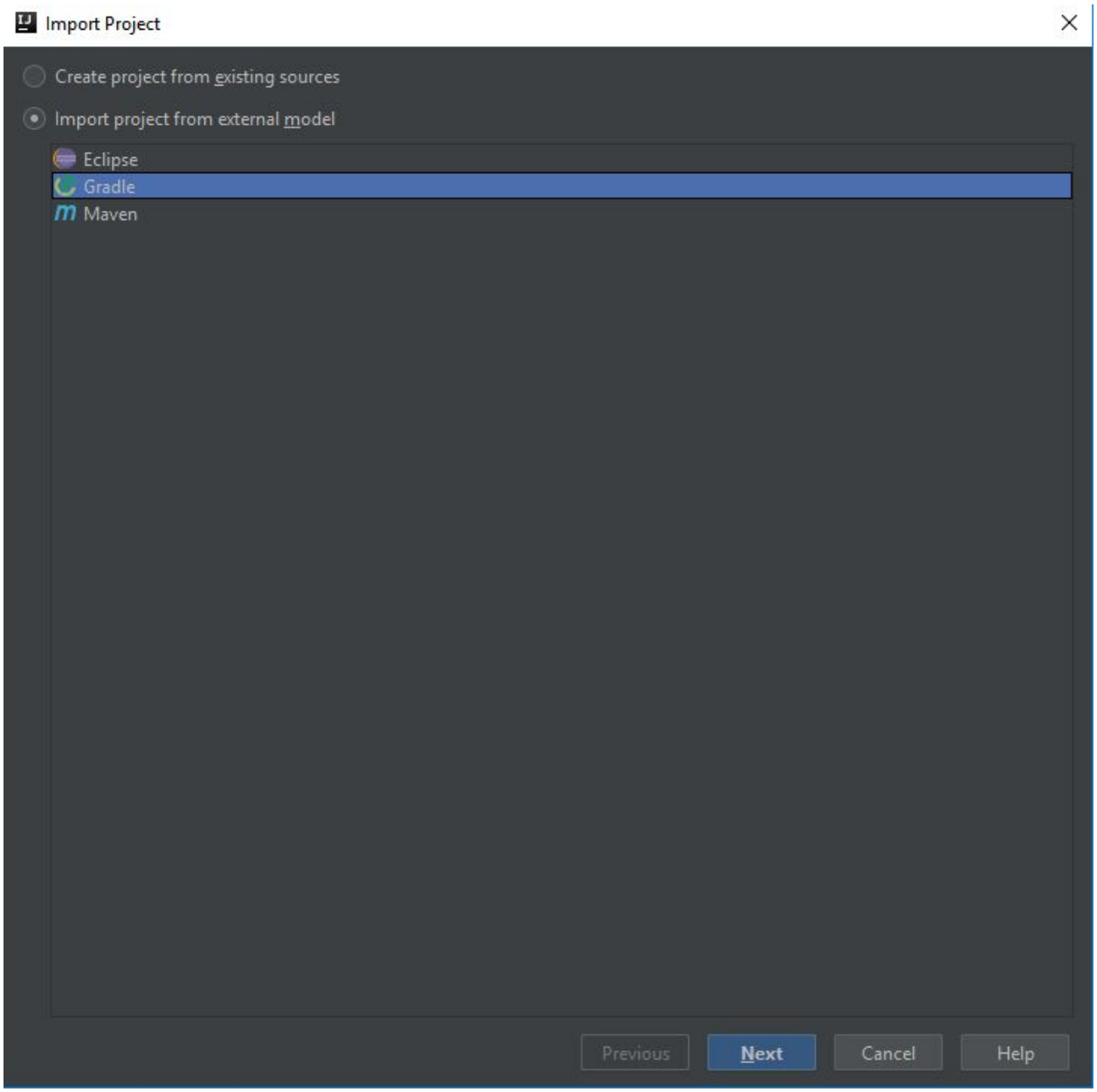
Abra o Intellij e selecione a opção “Import Project”.



Selecione a pasta “*cordapp-example*” e pressione OK.



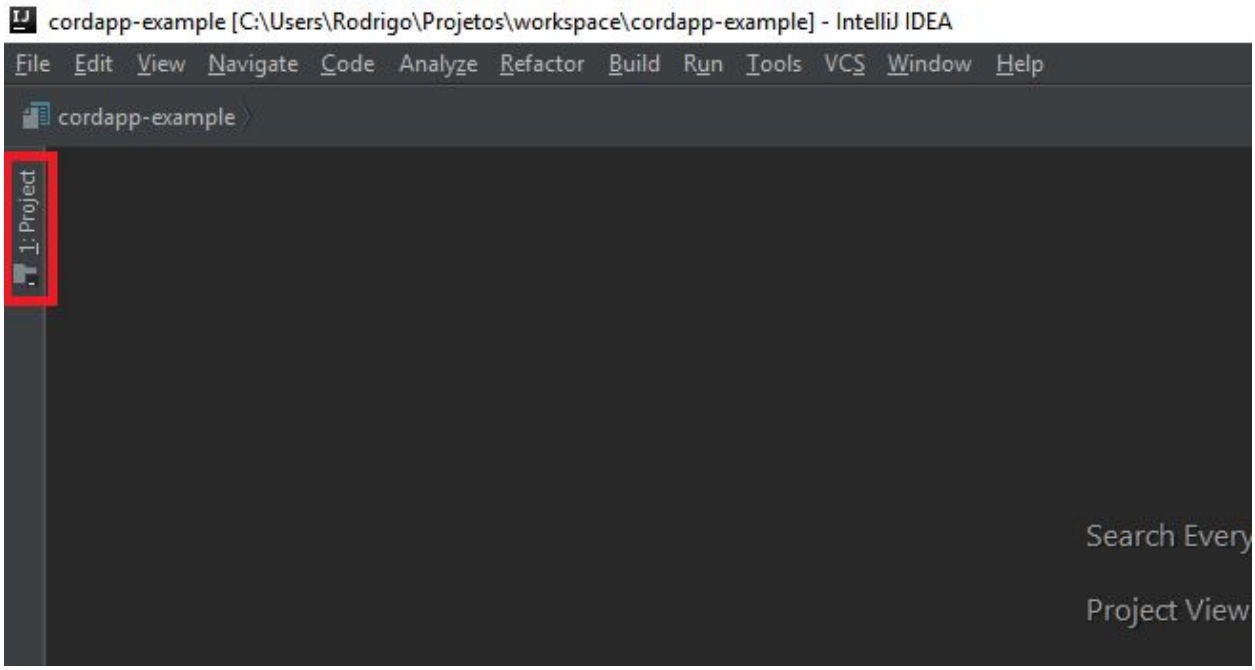
Selecione a opção “Gradle” e pressione “Next”.



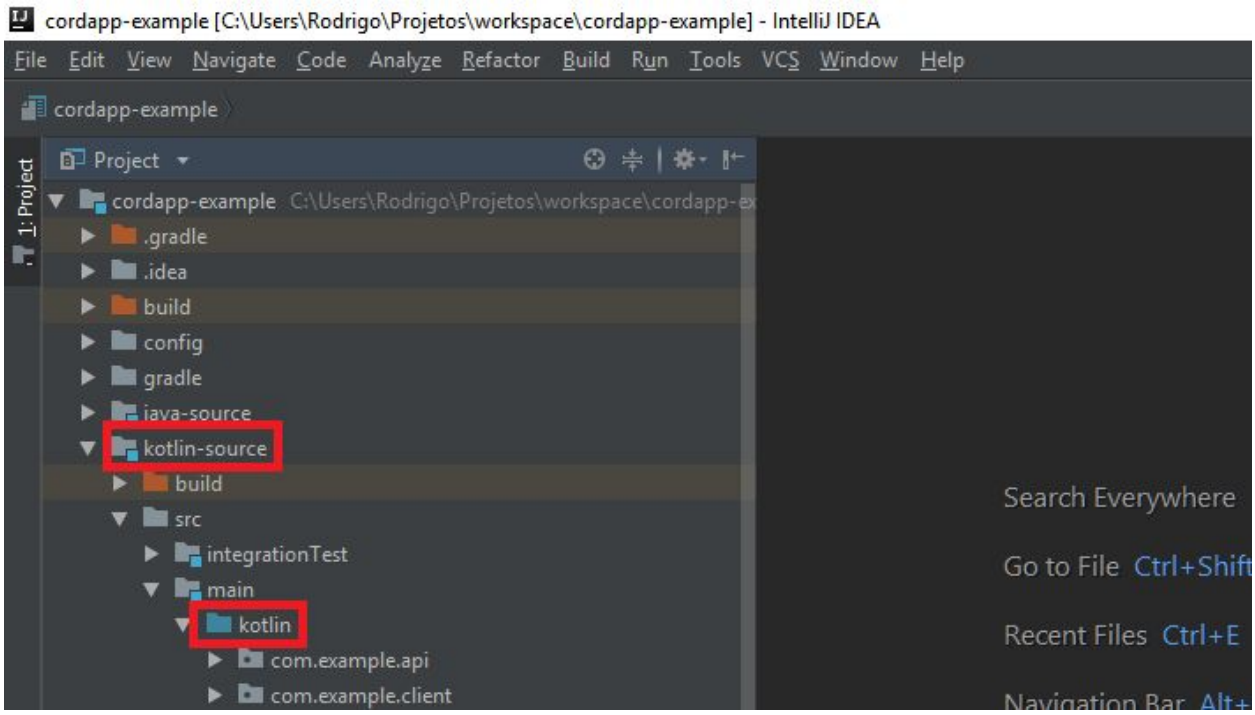
Após isso pressione “Finish”; caso ele solicite para sobrescrever o diretório “.idea” pressione “Yes”.

Ao abrir o projeto, um build do “*gradle*” será executado. Isto vai garantir que toda a indexação dos arquivos fique correta e vai facilitar a nossa navegação.

Acesse o menu de arquivos.



Navegue até a pasta “*kotlin*” dentro de “*kotlin-source*”.



Agora vamos dar uma olhada nas implementações.

Conceitos

Temos 3 peças fundamentais para a construção de um corDapp, os States, Contract e Flows. Cada um deles tem um papel fundamental na forma como a nossa aplicação irá executar.

State

O State representa o estado de um objeto qualquer em um momento do tempo, ou seja, o State representa todo o histórico de um objeto dentro da rede Corda.

No Corda, um State é representado por uma "*data class*", uma classe que possui apenas funções para manipular os seus próprios valores. O modificador "*data*" vai facilitar algumas implementações que vamos ver mais para a frente.

Acesse o pacote "*com.example.state*" e abra o arquivo "*IOUState*".

```
package com.example.state

import com.example.schema.IOUSchemaV1
import net.corda.core.contracts.ContractState
import net.corda.core.contracts.LinearState
import net.corda.core.contracts.UniqueIdentifier
import net.corda.core.identity.AbstractParty
import net.corda.core.identity.Party
import net.corda.core.schemas.MappedSchema
import net.corda.core.schemas.PersistentState
import net.corda.core.schemas.QueryableState
import java.time.Instant

/**
* The state object recording IOU agreements between two parties.
*
* A state must implement [ContractState] or one of its descendants.
*
* @param value the value of the IOU.
* @param lender the party issuing the IOU.
* @param borrower the party receiving and approving the IOU.
*/
data class IOUState(val value: Int,
                    val lender: Party,
                    val borrower: Party,
                    val dueDate: Instant,
                    val interest: Int,
                    val payment: Int = 0,
                    val status: String = "Created",
                    override val linearId: UniqueIdentifier = UniqueIdentifier()):
        LinearState, QueryableState {
    /** The public keys of the involved parties. */
```

```
    override val participants: List<AbstractParty> get() = listOf(lender, borrower)

    override fun generateMappedObject(schema: MappedSchema): PersistentState {
        return when (schema) {
            is IOUSchemaV1 -> IOUSchemaV1.PersistentIOU(
                    this.lender.name.toString(),
                    this.borrower.name.toString(),
                    this.value,
                    this.dueDate.toString(),
                    this.interest,
                    this.payment,
                    this.status,
                    this.linearId.id
            )
            else -> throw IllegalArgumentException("Unrecognised schema $schema")
        }
    }

    override fun supportedSchemas(): Iterable<MappedSchema> = listOf(IOUSchemaV1)
}
```

Algumas coisas que podemos observar sobre Kotlin:

- O construtor é escrito diretamente junto com a declaração da classe;
- As variáveis não precisam de **getter** e **setter**;
- Os valores declarados como **val** só podem receber uma atribuição, igual à um **final** no Java;
- Declara-se herança utilizando o operador “:”;
- Valores podem sofrer **override**.
- Funções são declaradas utilizando a palavra reservada **fun**.
- Os retornos das funções são declarados após o operador “:”;
- Não há diferenciação entre **Herança** e implementação de **Interface**;
- Suporte à **Pattern Matching**;
- Construção de objetos sem a utilização da palavra reservada **new**;

A classe IOUState herda da classe “Linear State” e utiliza a Interface “QueryableState”.

O “Linear State” representa um State que se comporta no modelo **UTXO**, ou seja, possui apenas uma única verdade em algum momento do tempo, e novas transações consomem as transações anteriores. Um exemplo de **UTXO** fora do Corda é o Bitcoin.

A Interface “QueryableState” irá nos permitir fazer queries customizadas sobre o Corda, mais sobre isso na sequência.

Contract

Um contract representa o que pode ou não ser feito com um ou mais States dentro de uma transação do Corda. Ele deve verificar as regras de negócio e validar que todas as informações foram preenchidas corretamente, de forma a evitar problemas de programação.

Ele deve representar um contrato físico em forma de código.

Diferente do State, um Contract não representa dados, e sim regras, por isso utiliza-se apenas "*class*" na sua definição.

No pacote "com.example.contract" acesse o arquivo "IOUContract".

```
package com.example.contract

import com.example.state.IOUState
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

/**
* A implementation of a basic smart contract in Corda.
*
* This contract enforces rules regarding the creation of a valid [IOUState], which in
turn encapsulates an [IOU].
*
* For a new [IOU] to be issued onto the ledger, a transaction is required which takes:
* - Zero input states.
* - One output state: the new [IOU].
* - An Create() command with the public keys of both the lender and the borrower.
*
* All contracts must sub-class the [Contract] interface.
*/
open class IOUContract : Contract {
   companion object {
       @JvmStatic
       val IOU_CONTRACT_ID = "com.example.contract.IOUContract"
   }

   /**
    * The verify() function of all the states' contracts must not throw an exception
for a transaction to be
    * considered valid.
    */
   override fun verify(tx: LedgerTransaction) {
       val command = tx.commands.requireSingleCommand<Commands.Create>()
       requireThat {
           // Generic constraints around the IOU transaction.
           "No inputs should be consumed when issuing an IOU." using
(tx.inputs.isEmpty())
           "Only one output state should be created." using (tx.outputs.size == 1)
           val out = tx.outputsOfType<IOUState>().single()
```

```
            "The lender and the borrower cannot be the same entity." using (out.lender
!= out.borrower)
            "All of the participants must be signers." using
(command.signers.containsAll(out.participants.map { it.owningKey }))

            // IOU-specific constraints.
            "The IOU's value must be non-negative." using (out.value > 0)
        }
    }

    /**
     * This contract only implements one command, Create.
     */
    interface Commands : CommandData {
        class Create : Commands
    }
}
```

Aqui temos mais algumas construções específicas do Kotlin:

- A palavra reservada **open** indica que a classe pode ser extendida, no Kotlin, por padrão a Herança não pode ser realizada.
- Temos também um **companion object.**Um companion object é uma instância que irá acompanhar a classe, contém métodos e variáveis consideradas estáticas, únicas para aquela classe e qualquer instância desta classe.
- Temos um bloco de função **requireThat**. O requireThat irá testar o valor que está presente após a palavra reservada **using**, caso o resultado seja falso, uma exceção será lançada com o texto definido antes da palavra reservada **using**.

A classe IOUContract herda diretamente da classe **Contract.** A classe Contract é a base para qualquer contrato escrito em Corda. Ao executar uma transação, será verificado todos os Contracts que estão presentes e será executado o método **verify** para garantir que não há inconsistência na transformação que foi realizada nos States desta transação.

Dentro do Corda, todas as transações carregam com elas, comandos que informam qual tipo de operação será realizada; podemos ter por exemplo, criação, exclusão e atualização de um dado. De acordo com o comando que será utilizado, teremos regras diferentes de validação.

Para indicar qual o comando que será utilizado, estendemos a interface “CommandData”, é a estrutura que fala para o Corda que as suas classes internas são os comando permitidos.

Neste exemplo, o único comando permitido é o comando “*Create*” que irá adicionar uma nova dívida na rede.

Para entender como iremos realizar a validação dos dados da transação, precisamos entender um pouco de como ela funciona.

Na transação, temos acesso a duas listas que representam as Entradas da Transação e as Saídas da Transação.

No Corda, uma Entrada em uma transação representa uma objeto que já está persistido no banco de dados, que já possui um estado inicial. Uma Saída representa uma novo Estado que está sendo escrito no banco de dados. Existem 3 combinações possíveis, que irão gerar resultados diferentes, são eles:

- Transação somente com Saída: Representa a criação de um State
- Transação com Entrada e Saída: Representa a atualização de um State
- Transação somente com Entrada: Representa a exclusão de um State

Verificação de assinaturas

Na linha :

```
"All of the participants must be signers." using
(command.signers.containsAll(out.participants.map { it.owningKey }))
```

Temos a verificação de quais são os participantes que devem fazer parte da transação.

Este passo é importante pois, como não temos consenso distribuído igual nas plataformas de blockchain, precisamos ter certeza que todas as partes envolvidas realmente concordaram em realizar a alteração e estão cientes que o dado já não está no mesmo estado, assim garantimos que todas as partes vão sempre estar enxergando a mesma verdade.

Flow

O Flow é a classe que descreve como as transações serão criadas e o que cada parte precisa fazer de ação dentro deste fluxo. Normalmente, é construído um ou mais flows para cada comando especificado no Contract.

Diferente do Contract e do State, não podemos ter mais de um Flow de um mesmo tipo, por isso utilizamos a estrutura “*object*” que no Kotlin representa um singleton, ou seja, uma classe que sempre que for chamada pelo seu nome, sempre retorna a mesma instância de objeto.

Abra o arquivo “ExampleFlow” no pacote “com.example.flow”.

```
package com.example.flow
```

```
import co.paralleluniverse.fibers.Suspendable
import com.example.contract.IOUContract
import com.example.contract.IOUContract.Companion.IOU_CONTRACT_ID
import com.example.flow.ExampleFlow.Acceptor
import com.example.flow.ExampleFlow.Initiator
import com.example.state.IOUState
import net.corda.core.contracts.Command
import net.corda.core.contracts.requireThat
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 net.corda.core.utilities.ProgressTracker.Step

/**
* This flow allows two parties (the [Initiator] and the [Acceptor]) to come to an
agreement about the IOU encapsulated
* within an [IOUState].
*
* In our simple example, the [Acceptor] always accepts a valid IOU.
*
* These flows have deliberately been implemented by using only the call() method for
ease of understanding. In
* practice we would recommend splitting up the various stages of the flow into
sub-routines.
*
* All methods called within the [FlowLogic] sub-class need to be annotated with the
@Suspendable annotation.
*/
object ExampleFlow {
   @InitiatingFlow
   @StartableByRPC
   class Initiator(val iouValue: Int,
                   val otherParty: Party) : FlowLogic<SignedTransaction>() {
       /**
        * The progress tracker checkpoints each stage of the flow and outputs the
specified messages when each
        * checkpoint is reached in the code. See the 'progressTracker.currentStep'
expressions within the call() function.
        */
       companion object {
           object GENERATING_TRANSACTION : Step("Generating transaction based on new
IOU.")
           object VERIFYING_TRANSACTION : Step("Verifying contract constraints.")
           object SIGNING_TRANSACTION : Step("Signing transaction with our private
key.")
           object GATHERING_SIGS : Step("Gathering the counterparty's signature.") {
               override fun childProgressTracker() = CollectSignaturesFlow.tracker()
           }
```

```
            object FINALISING_TRANSACTION : Step("Obtaining notary signature and
recording transaction.") {
                override fun childProgressTracker() = FinalityFlow.tracker()
            }

            fun tracker() = ProgressTracker(
                    GENERATING_TRANSACTION,
                    VERIFYING_TRANSACTION,
                    SIGNING_TRANSACTION,
                    GATHERING_SIGS,
                    FINALISING_TRANSACTION
            )
        }

        override val progressTracker = tracker()

        /**
         * The flow logic is encapsulated within the call() method.
         */
        @Suspendable
        override fun call(): SignedTransaction {
            // Obtain a reference to the notary we want to use.
            val notary = serviceHub.networkMapCache.notaryIdentities[0]

            // Stage 1.
            progressTracker.currentStep = GENERATING_TRANSACTION
            // Generate an unsigned transaction.
            val iouState = IOUState(iouValue,
serviceHub.myInfo.legalIdentities.first(), otherParty)
            val txCommand = Command(IOUContract.Commands.Create(),
iouState.participants.map { it.owningKey })
            val txBuilder = TransactionBuilder(notary)
                    .addOutputState(iouState, IOU_CONTRACT_ID)
                    .addCommand(txCommand)

            // Stage 2.
            progressTracker.currentStep = VERIFYING_TRANSACTION
            // Verify that the transaction is valid.
            txBuilder.verify(serviceHub)

            // Stage 3.
            progressTracker.currentStep = SIGNING_TRANSACTION
            // Sign the transaction.
            val partSignedTx = serviceHub.signInitialTransaction(txBuilder)

            // Stage 4.
            progressTracker.currentStep = GATHERING_SIGS
            // Send the state to the counterparty, and receive it back with their
signature.
```

```
            val otherPartyFlow = initiateFlow(otherParty)
            val fullySignedTx = subFlow(CollectSignaturesFlow(partSignedTx,
setOf(otherPartyFlow), GATHERING_SIGS.childProgressTracker()))

            // Stage 5.
            progressTracker.currentStep = FINALISING_TRANSACTION
            // Notarise and record the transaction in both parties' vaults.
            return subFlow(FinalityFlow(fullySignedTx,
FINALISING_TRANSACTION.childProgressTracker()))
        }
    }

    @InitiatedBy(Initiator::class)
    class Acceptor(val otherPartyFlow: FlowSession) : FlowLogic<SignedTransaction>() {
        @Suspendable
        override fun call(): SignedTransaction {
            val signTransactionFlow = object : SignTransactionFlow(otherPartyFlow) {
                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
                    "I won't accept IOUs with a value over 100." using (iou.value <=
100)
                }
            }

            return subFlow(signTransactionFlow)
        }
    }
}
```

A parte mais pesada do código se encontra nesta classe. Vamos primeiro dar uma olhada nas estruturas da classe:

- Temos um **object** que representa um flow, neste flow vamos descrever duas partes:
    - **Initiator**: O inicializador de um novo fluxo, descreve como será a operação por parte do inicializador.
    - **Acceptor**: O receptor de um fluxo, descreve como será a operação caso você receba uma chamada deste Flow.
- Temos um **companion object** que descreve os passos pelos quais iremos passar; além de conter os trackers de sub processos que serão executados.

Como pode ser observado, é necessário colocar uma anotação que demonstra qual classe é responsável pela inicialização do Flow (`@InitiatingFlow`), ela vai indicar quem é que está inicializando o Flow.

Para que este Flow seja controlado pelo Corda, é necessário herdar a classe base “FlowLogic<T>”. Um “FlowLogic”, é uma classe dentro do Corda que possui uma finalidade definida que ao ser chamada irá sempre retornar o objeto especificado do tipo T, que pode ser qualquer tipo de objeto. Neste exemplo, é retornada a transação para que possamos utilizar esta informação para saber se deu certo ou errado.

A transação que é retornada é uma classe do tipo “SignedTransaction” como resultado, ela representa uma transação que contém ao menos uma assinatura.

A classe base flow logic exige que seja implementado o método “call”, que não irá receber nenhuma entrada e deve retornar uma “SignedTransaction”.

É necessário informar que a função call pode ser suspendida em situações em que precisamos aguardar o retorno de algum outro node, utilizamos para isso a anotação “`@Suspendable`”. Para realizar esta suspensão, o Corda utiliza co-rotinas, para saber mais sobre co-rotinas acesse: https://en.wikipedia.org/wiki/Coroutine e https://github.com/puniverse/quasar

O primeiro passo é escolher qual vai ser o Notary que irá validar a nossa transação. Para isto, vamos utilizar o serviceHub.

O Corda oferece diversos serviços dentro do seu node através do objeto serviceHub, que está criado na classe base “FlowLogic” e conseguimos acessar em nossa função. Como estamos trabalhando com um exemplo, o Notary que vamos utilizar será o primeiro que encontramos na rede.

Após a seleção do Notary, é necessário construir o State que será transacionado e construir o comando que irá informar qual a intenção desta transação.

Para a seleção dos participantes, estamos informando apenas a chave pública, desta forma, apenas a chave pública é enviada para os nós envolvidos na transação, desta forma, a transação é pseudo-anônima entre os participantes, apenas quem conhece a chave pública da outra entidade vai reconhecer o seu dono.

Com a construção do State e do Command, já é possível construir a transação. Para facilitar a construção da transação, temos o “TransactionBuilder”, classe que recebe todos os parâmetros necessários para construir uma transação e realiza todo o trabalho de garantir que a transação vai ser legível para as outras partes.

Uma vez construída a transação, é necessário verificar se tudo está OK, chamando o método “*verify*” na transação, todos os contratos serão executados, garantindo que não cometemos nenhum engano no momento de construir os objetos.

Após a verificação, o passo seguinte é a assinatura, todo o processo de assinatura digital é abstraída no metodo “signInitialTransaction” dentro do “serviceHub”.

Agora é necessário enviar a transação para a outra parte para que ela verifique e assine a transação. Como a coleta de assinatura é um processo padrão, já existe um Flow no Corda que cuida disso, o “CollectSignaturesFlow”, que irá enviar para todos os nós tomarem uma decisão sobre o contrato e então devolver a transação com a sua assinatura.

Uma vez assinada pelas duas partes, utilizamos um Flow já construído na plataforma para realizar a persistência do dado, o “FinalityFlow”. O FinalityFlow irá verificar se a transação possui todas as assinaturas necessárias, irá enviar a transação para o Notary desta transação também executar os contratos (caso esteja marcado como **validanting: true**) e assinar a transação e, após a validação do Notary, enviar para todas as partes envolvidas o sinal de que é seguro concretizar a transação e alterar o seu banco de dados.

Corda RPC

Conhecendo as estruturas básicas necessárias para o funcionamento do Corda, agora vamos olhar a forma como conseguimos conversar com o nosso Nó.

Toda comunicação com o Corda é realizado via AMQP 1.0/TLS, porém, para evitar que existam problemas na forma como são montados as mensagens, é oferecido o pacote “net.corda.client.rpc.CordaRPCClient”, que abstraí a mensageria e entrega métodos de alto nível para controle e manipulação do nó.

Acesse a classe “ExampleClientRPC” no pacote “com.example.client”.

```
package com.example.client

import com.example.state.IOUState
import net.corda.client.rpc.CordaRPCClient
import net.corda.core.contracts.StateAndRef
import net.corda.core.utilities.NetworkHostAndPort
import net.corda.core.utilities.loggerFor
import org.slf4j.Logger

/**
*  Demonstration of using the CordaRPCClient to connect to a Corda Node and
*  steam some State data from the node.
**/

fun main(args: Array<String>) {
   ExampleClientRPC().main(args)
}
```

```
private class ExampleClientRPC {
   companion object {
       val logger: Logger = loggerFor<ExampleClientRPC>()
       private fun logState(state: StateAndRef<IOUState>) = logger.info("{}",
state.state.data)
   }

   fun main(args: Array<String>) {
       require(args.size == 1) { "Usage: ExampleClientRPC <node address>" }
       val nodeAddress = NetworkHostAndPort.parse(args[0])
       val client = CordaRPCClient(nodeAddress)

       // Can be amended in the com.example.MainKt file.
       val proxy = client.start("user1", "test").proxy

       // Grab all existing and future IOU states in the vault.
       val (snapshot, updates) = proxy.vaultTrack(IOUState::class.java)

       // Log the 'placed' IOU states and listen for new ones.
       snapshot.states.forEach { logState(it) }
       updates.toBlocking().subscribe { update ->
           update.produced.forEach { logState(it) }
       }
   }
}
```

Neste pequeno exemplo, podemos ver como é feita a subscriação à eventos de alteração nos dados. Primeiro, é necessário iniciar um clientRPC, para isso precisamos passar um usuário e senha configurado no arquivo “node.conf”. Após isso podemos acessar vários métodos através do **proxy** que será disponibilizado. Aqui é feita uma subscrição que seja logado todas as alterações que forem feitos no Vault em relação à classe IOUState.

Na próxima etapa, vamos dar uma olhada na forma como conseguimos iniciar flows.

Corda APIs

Uma das formas mais simples de prototipar os nossos serviços é construindo APIs e utilizando o corda-webserver.

As APIs em Corda devem seguir o modelo JSR 311 : https://jcp.org/en/jsr/detail?id=311

Acesse o arquivo “ExampleAPI.kt” no pacote “com.example.api”.

```
package com.example.api

import com.example.flow.ExampleFlow.Initiator
import com.example.schema.IOUSchemaV1
import com.example.state.IOUState
```

```
import net.corda.core.identity.CordaX500Name
import net.corda.core.messaging.CordaRPCOps
import net.corda.core.messaging.startTrackedFlow
import net.corda.core.messaging.vaultQueryBy
import net.corda.core.node.services.IdentityService
import net.corda.core.node.services.Vault
import net.corda.core.node.services.vault.QueryCriteria
import net.corda.core.node.services.vault.builder
import net.corda.core.utilities.getOrThrow
import net.corda.core.utilities.loggerFor
import org.slf4j.Logger
import javax.ws.rs.*
import javax.ws.rs.core.MediaType
import javax.ws.rs.core.Response
import javax.ws.rs.core.Response.Status.BAD_REQUEST
import javax.ws.rs.core.Response.Status.CREATED

val SERVICE_NAMES = listOf("Notary", "Network Map Service")

// This API is accessible from /api/example. All paths specified below are relative to
it.
@Path("example")
class ExampleApi(private val rpcOps: CordaRPCOps) {
   private val myLegalName: CordaX500Name =
rpcOps.nodeInfo().legalIdentities.first().name

   companion object {
       private val logger: Logger = loggerFor<ExampleApi>()
   }

   /**
    * Returns the node's name.
    */
   @GET
   @Path("me")
   @Produces(MediaType.APPLICATION_JSON)
   fun whoami() = mapOf("me" to myLegalName)

   /**
    * Returns all parties registered with the [NetworkMapService]. These names can be
used to look up identities
    * using the [IdentityService].
    */
   @GET
   @Path("peers")
   @Produces(MediaType.APPLICATION_JSON)
   fun getPeers(): Map<String, List<CordaX500Name>> {
       val nodeInfo = rpcOps.networkMapSnapshot()
       return mapOf("peers" to nodeInfo
               .map { it.legalIdentities.first().name }
```

```
                //filter out myself, notary and eventual network map started by driver
                .filter { it.organisation !in (SERVICE_NAMES +
myLegalName.organisation) })
    }

    /**
     * Displays all IOU states that exist in the node's vault.
     */
    @GET
    @Path("ious")
    @Produces(MediaType.APPLICATION_JSON)
    fun getIOUs() = rpcOps.vaultQueryBy<IOUState>().states

    /**
     * Initiates a flow to agree an IOU between two parties.
     *
     * Once the flow finishes it will have written the IOU to ledger. Both the lender
and the borrower will be able to
     * see it when calling /api/example/ious on their respective nodes.
     *
     * This end-point takes a Party name parameter as part of the path. If the serving
node can't find the other party
     * in its network map cache, it will return an HTTP bad request.
     *
     * The flow is invoked asynchronously. It returns a future when the flow's call()
method returns.
     */
    @PUT
    @Path("create-iou")
    fun createIOU(@QueryParam("iouValue") iouValue: Int, @QueryParam("partyName")
partyName: CordaX500Name?): Response {
        if (iouValue <= 0 ) {
            return Response.status(BAD_REQUEST).entity("Query parameter 'iouValue' must
be non-negative.\n").build()
        }
        if (partyName == null) {
            return Response.status(BAD_REQUEST).entity("Query parameter 'partyName'
missing or has wrong format.\n").build()
        }
        val otherParty = rpcOps.wellKnownPartyFromX500Name(partyName) ?:
                return Response.status(BAD_REQUEST).entity("Party named $partyName
cannot be found.\n").build()

        return try {
            val signedTx = rpcOps.startTrackedFlow(::Initiator, iouValue,
otherParty).returnValue.getOrThrow()
            Response.status(CREATED).entity("Transaction id ${signedTx.id} committed to
ledger.\n").build()

        } catch (ex: Throwable) {
```

```
            logger.error(ex.message, ex)
            Response.status(BAD_REQUEST).entity(ex.message!!).build()
        }
    }

/**
     * Displays all IOU states that are created by Party.
     */
    @GET
    @Path("my-ious")
    @Produces(MediaType.APPLICATION_JSON)
    fun myious(): Response {
        val generalCriteria = QueryCriteria.VaultQueryCriteria(Vault.StateStatus.ALL)
        val results = builder {
                var partyType =
IOUSchemaV1.PersistentIOU::lenderName.equal(rpcOps.nodeInfo().legalIdentities.first().
name.toString())
                val customCriteria = QueryCriteria.VaultCustomQueryCriteria(partyType)
                val criteria = generalCriteria.and(customCriteria)
                val results = rpcOps.vaultQueryBy<IOUState>(criteria).states
                return Response.ok(results).build()
        }
    }
}
```

Aqui vamos conseguir visualizar todos os métodos disponibilizados via API, que verificamos na aula1.

Os métodos mais importantes para verificarmos são:

- **create-iou**
- **ious**
- **my-ious**

Nestes métodos vamos conseguir ver como são inicializados os Flows e como conseguimos consumir informação de nossa base de dados.

create-iou

O método **create-iou** recebe por parâmetro o valor da dívida que vai ser criada e com quem você tem esta dívida. Após algumas validações básicas, é necessário verificar se o nome recebido pela outra parte representa um nó dentro da rede. Para realizar esta operação, utilizamos o nosso **proxy de RPC** que desta vez recebemos como parâmetro no construtor de nossa API. Nele temos o método “wellKnownPartyForX500Name” que retorna uma entidade Party caso este nome esteja na lista de nós conhecidos ou nulo.

Para tratar o nulo, temos um operador especial do Kotlin, o elvis operator "**?:**", caso o valor seja nulo, ele irá executar a função que está após o operador, do contrário ele garante que o objeto não está nulo, permitindo acesso seguro ao objeto.

Agora que já temos a referência correta para o outro nó, conseguimos criar a dívida executando o flow ExampleFlow. Para fazer isso, utilizamos outro método presente no nosso **proxy de RPC** o método "startTrackedFlow". Este método solicita a classe que deve ser inicializada e quais são os parâmetros que devem ser utilizados no construtor. Ao executar este comando, será retornado um objeto do tipo "Future", que irá indicar quando o processo foi finalizado, já que a operação é totalmente assíncrona.

ious

O método **ious** irá retornar todos as dívidas ativas que são conhecidas por este nó, sejam eles dívidas dele ou para ele.

Para fazer o acesso, é utilizado o serviço "Vault". O "Vault" é a nossa base de dados, onde temos todas as informações de todos os states e transações que ocorrem neste nó.

O método utilizado é o "*vaultQueryBy*", que recebe qual a classe de State que estamos buscando. Como nenhum critério de seleção foi informado, ele irá retornar todos os IOUStates não consumidos.

my-ious

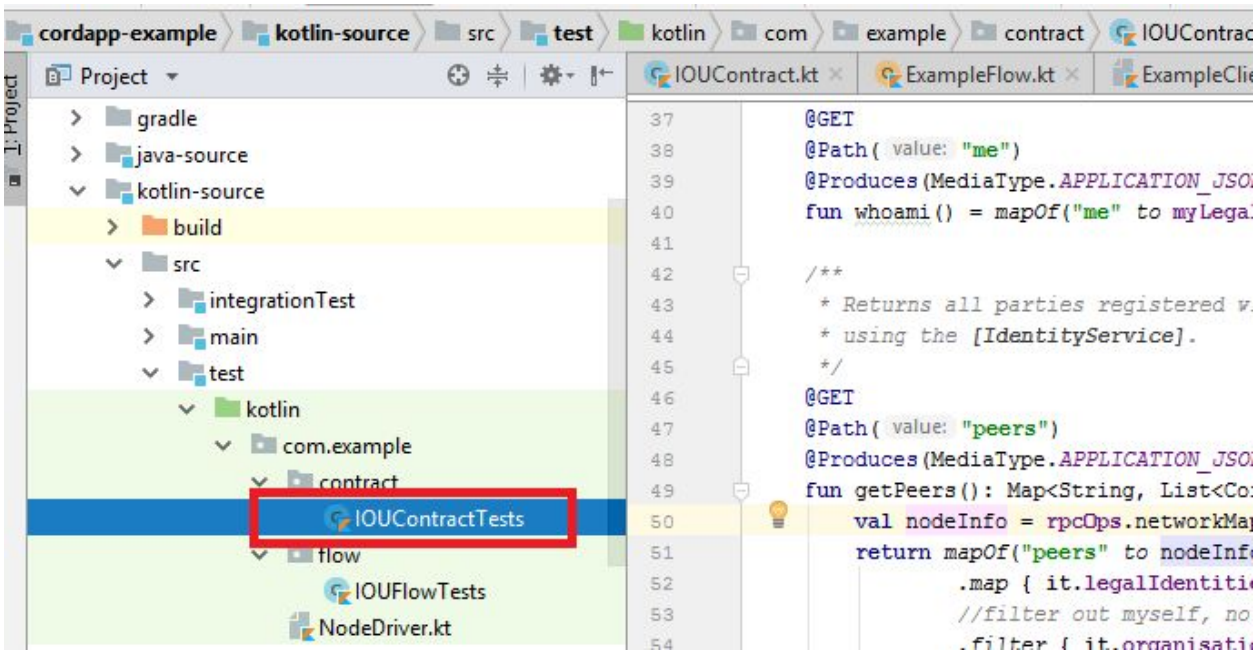
O método **my-ious** irá retornar todos o histórico relacionado à dívidas onde eu sou a parte que fez o empréstimo.

Para isso, é necessário fazer uma query customizada, para conseguir isto, utilizamos o bloco "*builder*". O "*builder*" irá adicionar os métodos de query sobre os para os atributos do nosso State, como por example o método "*equal*" que está sendo utilizado para comparar se o nome de quem cedeu o empréstimo é o nome deste nó.

Para mais informações de como construir queries no vault acesse: https://docs.corda.net/api-vault-query.html

Testes

No Corda, temos um framework construído para realização de testes integrados, no explorador de arquivos do projeto, fecha a pasta "main" e vá para a pasta "test", nela acessa o pacote "com.example.contract" e abra o arquivo "IOUContractTests".




```
package com.example.contract

import com.example.contract.IOUContract.Companion.IOU_CONTRACT_ID
import com.example.state.IOUState
import net.corda.core.identity.CordaX500Name
import net.corda.testing.core.TestIdentity
import net.corda.testing.node.MockServices
import net.corda.testing.node.ledger
import org.junit.Test

class IOUContractTests {
    private val ledgerServices = MockServices()
    private val megaCorp = TestIdentity(CordaX500Name("MegaCorp", "London", "GB"))
    private val miniCorp = TestIdentity(CordaX500Name("MiniCorp", "New York", "US"))

    @Test
    fun `transaction must include Create command`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                fails()
                command(listOf(megaCorp.publicKey, miniCorp.publicKey),
IOUContract.Commands.Create())
                verifies()
            }
        }
    }

    @Test
```

```
    fun `transaction must have no inputs`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                input(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                command(listOf(megaCorp.publicKey, miniCorp.publicKey),
IOUContract.Commands.Create())
                `fails with`("No inputs should be consumed when issuing an IOU.")
            }
        }
    }

    @Test
    fun `transaction must have one output`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                command(listOf(megaCorp.publicKey, miniCorp.publicKey),
IOUContract.Commands.Create())
                `fails with`("Only one output state should be created.")
            }
        }
    }

    @Test
    fun `lender must sign transaction`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                command(miniCorp.publicKey, IOUContract.Commands.Create())
                `fails with`("All of the participants must be signers.")
            }
        }
    }

    @Test
    fun `borrower must sign transaction`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                command(megaCorp.publicKey, IOUContract.Commands.Create())
                `fails with`("All of the participants must be signers.")
            }
        }
    }
```

```
    @Test
    fun `lender is not borrower`() {
        val iou = 1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, megaCorp.party, megaCorp.party))
                command(listOf(megaCorp.publicKey, miniCorp.publicKey),
IOUContract.Commands.Create())
                `fails with`("The lender and the borrower cannot be the same entity.")
            }
        }
    }

    @Test
    fun `cannot create negative-value IOUs`() {
        val iou = -1
        ledgerServices.ledger {
            transaction {
                output(IOU_CONTRACT_ID, IOUState(iou, miniCorp.party, megaCorp.party))
                command(listOf(megaCorp.publicKey, miniCorp.publicKey),
IOUContract.Commands.Create())
                `fails with`("The IOU's value must be non-negative.")
            }
        }
    }
}
```

No pacote “net.corda.testing” temos acesso a serviços mockados que irão simular o comportamento de um nó, para testar um contrato vamos utilizar o MockServices, que irá permitir a criação de transações, de forma descritiva, e verificar a sua validade, além de ser possível testar por erros utilizando o comando `*fails with*`.

Acesse o pacote “com.example.flow” e abra o arquivo “IOUFlowTests”.

```
package com.example.flow

import com.example.state.IOUState
import net.corda.core.contracts.TransactionVerificationException
import net.corda.core.node.services.queryBy
import net.corda.core.utilities.getOrThrow
import net.corda.testing.core.singleIdentity
import net.corda.testing.node.MockNetwork
import net.corda.testing.node.StartedMockNode
import org.junit.After
import org.junit.Before
import org.junit.Test
import kotlin.test.assertEquals
import kotlin.test.assertFailsWith
```

```
class IOUFlowTests {
    lateinit var network: MockNetwork
    lateinit var a: StartedMockNode
    lateinit var b: StartedMockNode

    @Before
    fun setup() {
        network = MockNetwork(listOf("com.example.contract"))
        a = network.createPartyNode()
        b = network.createPartyNode()
        // For real nodes this happens automatically, but we have to manually register
the flow for tests.
        listOf(a, b).forEach {
it.registerInitiatedFlow(ExampleFlow.Acceptor::class.java) }
        network.runNetwork()
    }

    @After
    fun tearDown() {
        network.stopNodes()
    }

    @Test
    fun `flow rejects invalid IOUs`() {
        val flow = ExampleFlow.Initiator(-1, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()

        // The IOUContract specifies that IOUs cannot have negative values.
        assertFailsWith<TransactionVerificationException> { future.getOrThrow() }
    }

    @Test
    fun `SignedTransaction returned by the flow is signed by the initiator`() {
        val flow = ExampleFlow.Initiator(1, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()

        val signedTx = future.getOrThrow()
        signedTx.verifySignaturesExcept(b.info.singleIdentity().owningKey)
    }

    @Test
    fun `SignedTransaction returned by the flow is signed by the acceptor`() {
        val flow = ExampleFlow.Initiator(1, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()

        val signedTx = future.getOrThrow()
```

```
        signedTx.verifySignaturesExcept(a.info.singleIdentity().owningKey)
    }

    @Test
    fun `flow records a transaction in both parties' transaction storages`() {
        val flow = ExampleFlow.Initiator(1, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()
        val signedTx = future.getOrThrow()

        // We check the recorded transaction in both transaction storages.
        for (node in listOf(a, b)) {
            assertEquals(signedTx,
node.services.validatedTransactions.getTransaction(signedTx.id))
        }
    }

    @Test
    fun `recorded transaction has no inputs and a single output, the input IOU`() {
        val iouValue = 1
        val flow = ExampleFlow.Initiator(iouValue, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()
        val signedTx = future.getOrThrow()

        // We check the recorded transaction in both vaults.
        for (node in listOf(a, b)) {
            val recordedTx =
node.services.validatedTransactions.getTransaction(signedTx.id)
            val txOutputs = recordedTx!!.tx.outputs
            assert(txOutputs.size == 1)

            val recordedState = txOutputs[0].data as IOUState
            assertEquals(recordedState.value, iouValue)
            assertEquals(recordedState.lender, a.info.singleIdentity())
            assertEquals(recordedState.borrower, b.info.singleIdentity())
        }
    }

    @Test
    fun `flow records the correct IOU in both parties' vaults`() {
        val iouValue = 1
        val flow = ExampleFlow.Initiator(1, b.info.singleIdentity())
        val future = a.startFlow(flow)
        network.runNetwork()
        future.getOrThrow()

        // We check the recorded IOU in both vaults.
        for (node in listOf(a, b)) {
            node.transaction {
```

```
                val ious = node.services.vaultService.queryBy<IOUState>().states
                assertEquals(1, ious.size)
                val recordedState = ious.single().state.data
                assertEquals(recordedState.value, iouValue)
                assertEquals(recordedState.lender, a.info.singleIdentity())
                assertEquals(recordedState.borrower, b.info.singleIdentity())
            }
        }
    }
}
```

Diferente do teste do contrato, para o flow, precisamos simular com o mínimo de nós suficientes para simular o funcionamento do nosso flow, para isso, vamos usar nós mockados. Utilizamos uma “MockNetwork” e dela conseguimos os nossos nós. Está é uma rede de testes sincrona, então para garantir que as operações foram realizadas, é necessário que executar o comando “*runNetwork*”. Aqui conseguimos ver que é possível acessar diretamente os métodos do nó além de ser possível acessar os dados do seu Vault.