Suporte ao desenvolvimento

Curso Blockchain Developer - Turma JUN2018

05de Julho de 2018

Material produzido por bbchain.

Projeto 1 - Aula 3 - corDapp IOU

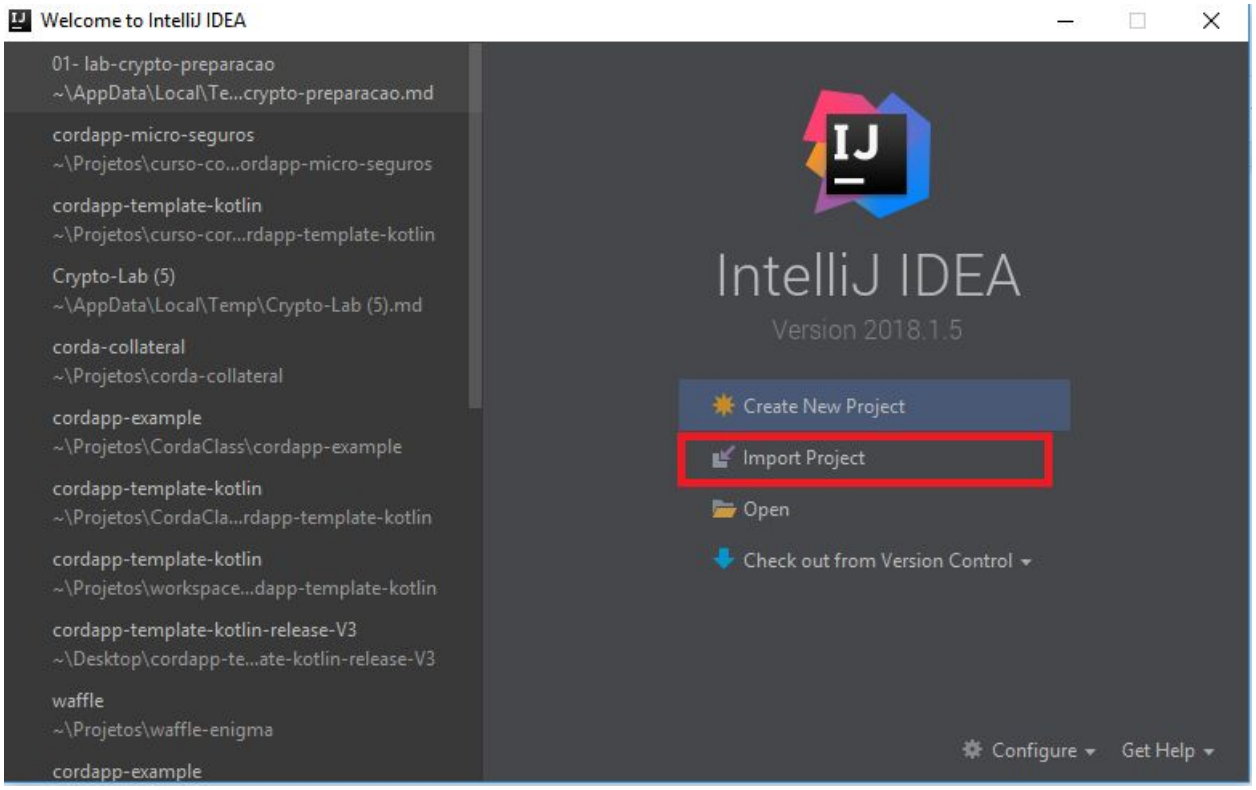
Nesta aula, vamos estender o projeto IOU adicionando a funcionalidade de pagamento. Vamos construir :

- State
- Contract
- Flow
- Testes
- API

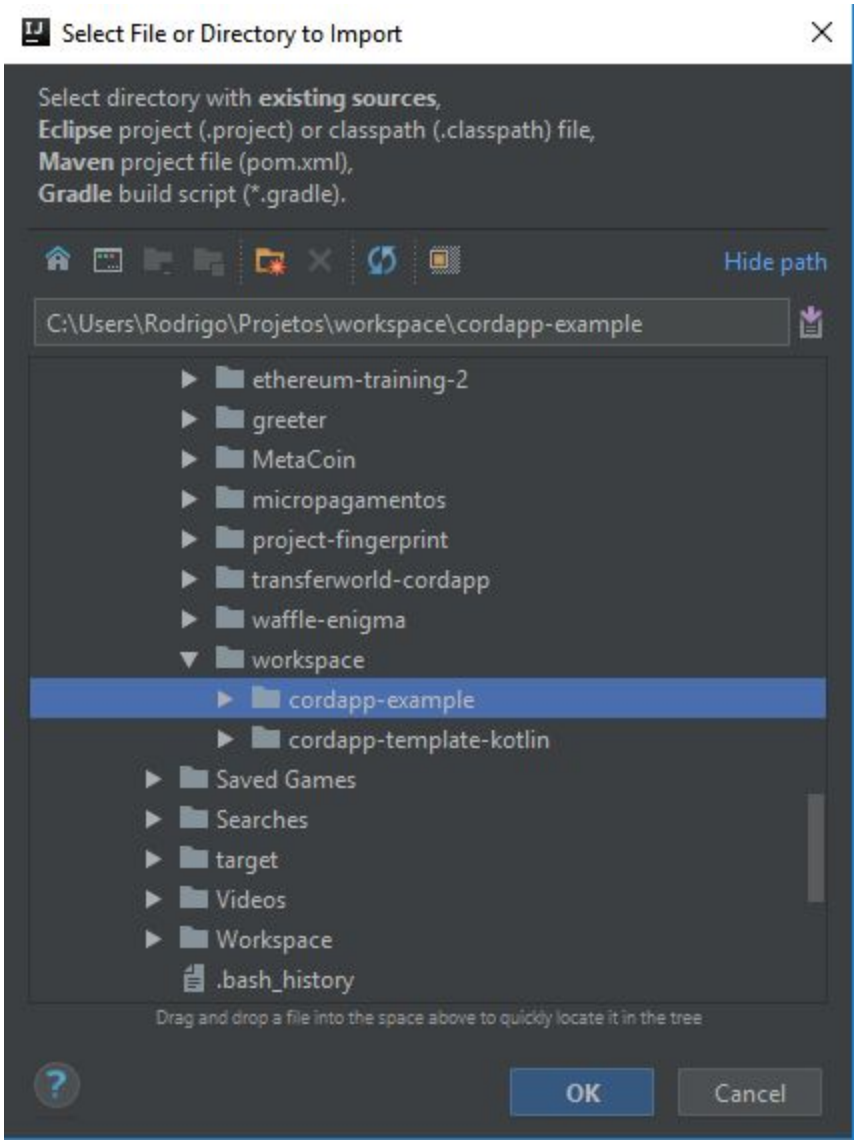
Para quem ainda não conseguiu abrir o projeto siga os passos abaixo:

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

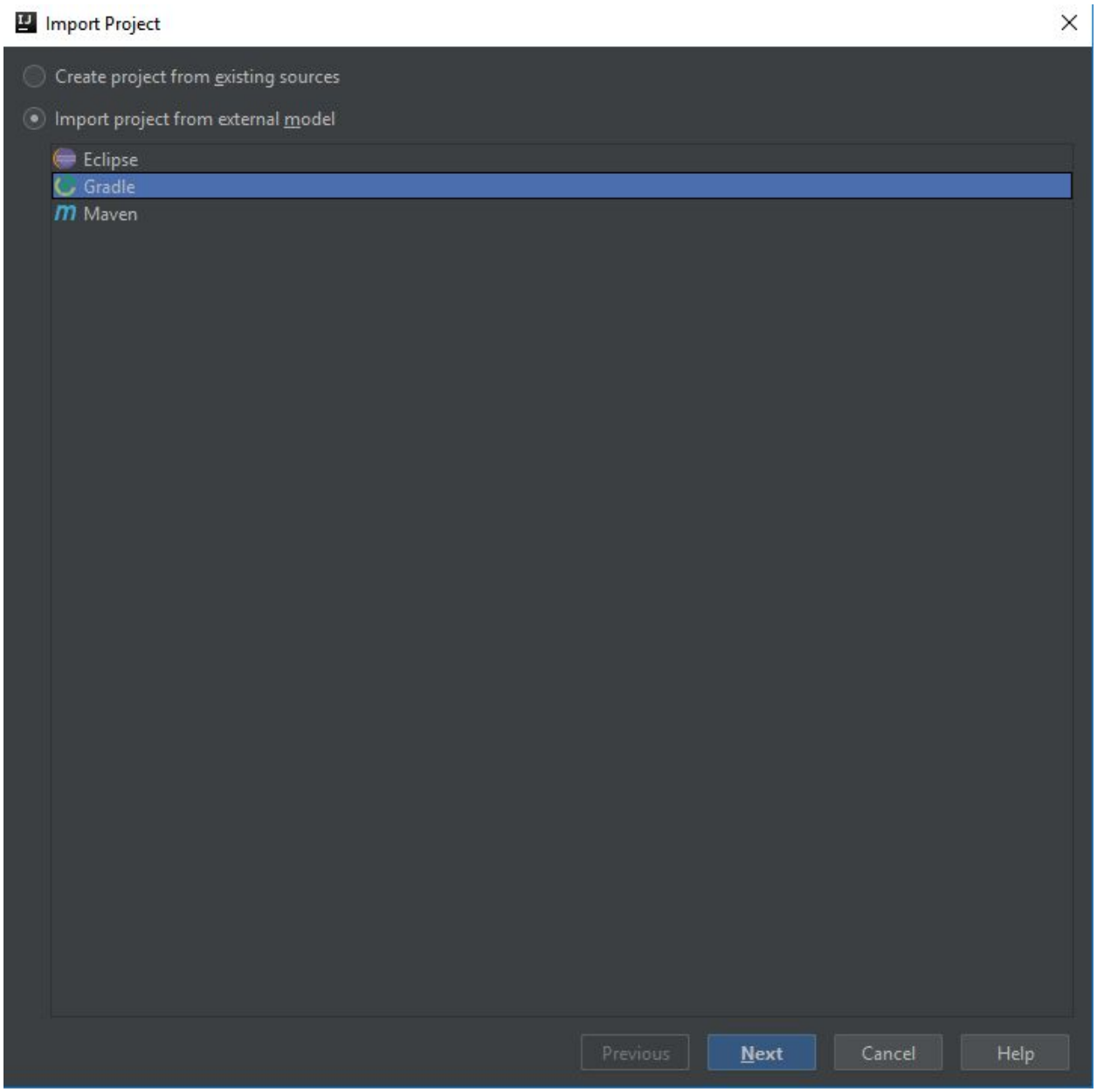
Abra o Intellij e selecione a opção “Import Project”.



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



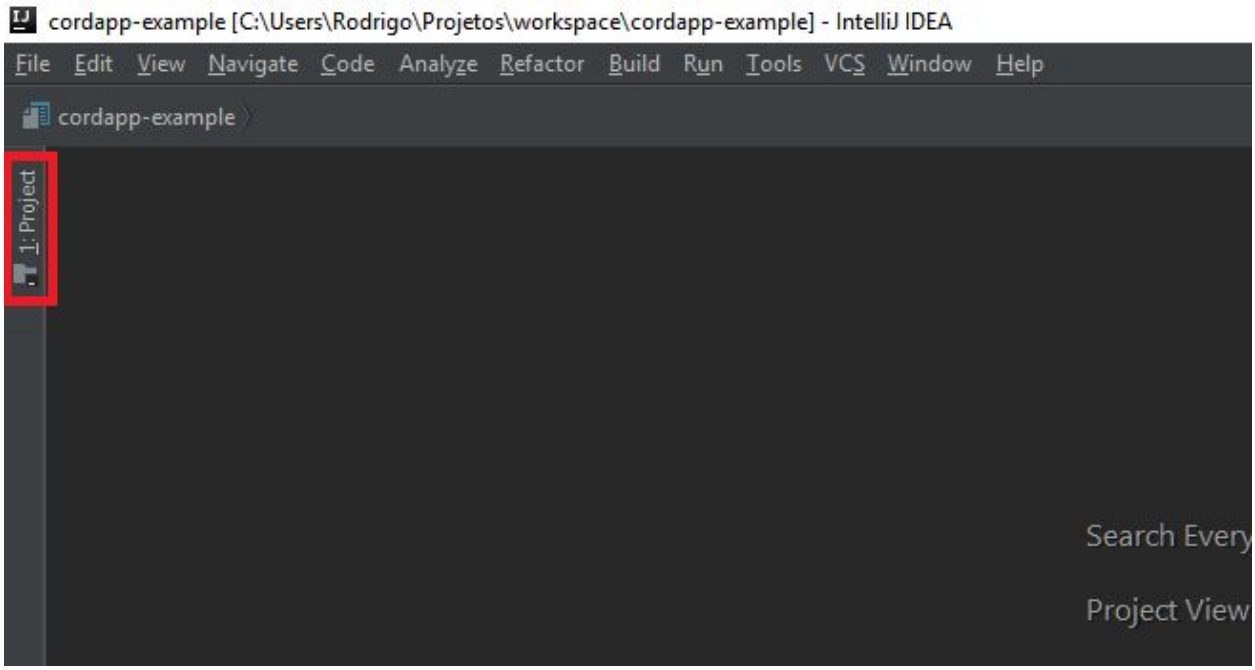
Selecione a opção “Gradle” e pressione “Next”.



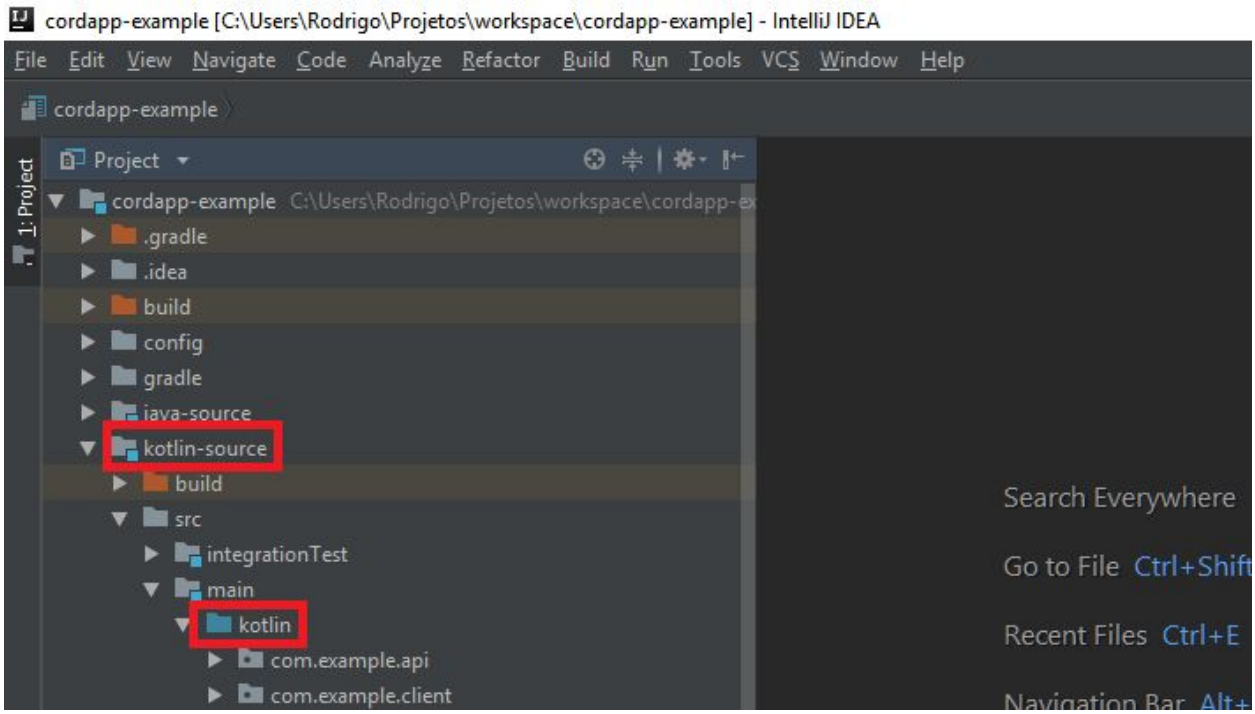
Após isso pressione “Finish”; caso ele solicite para sobrescrever o diretório “.idea” pressione “Yes”.

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*”.



State

Para a funcionalidade de pagamentos, vamos precisar adicionar novas informações no nosso State. Atualmente as únicas informações que temos é a do valor emprestado, quem está emprestando e quem pegou emprestado.

Vamos adicionar um novo atributo com a informação do valor que já foi pago, permitindo assim, pagamentos parciais.

Crie um campo do tipo **Int** com o nome **paymentValue** na classe **IOUState**.

```
        data class IOUState(val value: Int,
                            val lender: Party,
                            val borrower: Party,
                            val paymentValue: Int = 0,
                            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.linearId.id
                    )
                    else -> throw IllegalArgumentException("Unrecognised schema
$schema")
                }
            }

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

Contract

Para o contrato precisamos fazer algumas alterações, a primeira é verificar se o valor enviado no paymentValue está zerado durante a criação, já que, neste cenário, não será permitido nenhum pagamento na criação da dívida.

```
"The IOU's payment value must be 0." using (out.paymentValue == 0)
```

Agora que temos certeza que as regras de criação da dívida estão corretas, vamos adicionar um novo comando à lista de comandos, declarando que temos a intenção de fazer um pagamento da dívida.

```
interface Commands : CommandData {
   class Create : Commands
   class Pay: Commands
}
```

Com isto, agora conseguimos diferenciar quando o contrato está sendo usando para criar uma dívida ou pagar uma dívida. Para fazer isto, precisamos verificar o tipo do comando recebido. Na implementação atual, é verificado se o comando recebido é um comando de Create:

```
val command = tx.commands.requireSingleCommand<Commands.Create>()
```

Conseguimos fazer uma pequena alteração neste código para que seja requerido qualquer um dos nossos comandos, basta verificar apenas se ele é um “**Commands**”.

```
val command = tx.commands.requireSingleCommand<Commands>()
```

Desta forma, independente do comando ser “Create” ou “Pay”, o método “*requireSingleCommand*”, irá retornar o comando. Agora só precisamos identificar quando devemos aplicar as regras para o comando “Create” e quando aplicar para o comando “Pay”, para fazer isso vamos utilizar Pattern Matching.

```
when(command.value) {
   is Commands.Create -> TODO()
   is Commands.Pay -> TODO()
}
```

Com este comando conseguimos separar o que vai ser executado caso o que for recebido for um “Create” ou for um “Pay”. Agora só precisamos implementar as validações.

Para o “Create” já temos todas as validações prontas então, apenas para deixar o código mais organizado, vamos criar uma função que irá fazer a verificação do comando “Create”.

```
        override fun verify(tx: LedgerTransaction) {
            val command = tx.commands.requireSingleCommand<Commands>()
            when(command.value) {
                is Commands.Create -> verifyCreate(tx)
                is Commands.Pay -> TODO()
            }
        }

        private fun verifyCreate(tx: LedgerTransaction) {
            val command = tx.commands.requireSingleCommand<Commands>()
            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)
                "The IOU's payment value must be 0." using (out.paymentValue == 0)
            }
        }
```

Vamos agora fazer também a validação para o comando “Pay”.

O Comando “Pay” possui algumas características diferentes do comando “Create”, por ser um comando de atualização, precisamos ter tanto um State de Input, quanto de Output.

```
"Only one input should be consumed when paying an IOU." using (tx.inputs.size == 1)
"Only one output state should be created." using (tx.outputs.size == 1)
```

Outra coisa que precisamos verificar é se o Input e o Output são o mesmo State, já que é possível enviar qualquer State em uma transação.

```
                val input = tx.inputsOfType<IOUState>().single()
                val output = tx.outputsOfType<IOUState>().single()
                "The input and output state should be the same." using
                        (input.linearId == output.linearId)
```

A validação das assinaturas também tem que continuar existindo, já que queremos que as duas partes concordem com a alteração.

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

Agora que já validamos a entrada dos dados, precisamos verificar as regras de negócio. Neste comando vamos permitir pagamentos parciais, então conseguimos identificar duas regras.

- O valor de pagamento no Output tem que ser maior que o valor de pagamento no Input, garantindo que algum valor a mais foi pago com esta transação.
- O valor de pagamento no Output não pode ser maior que o valor do empréstimo, garantindo que não vai existir over pay.

Implementando elas temos:

```
"O valor de pagamento no Output tem que ser maior que o valor de pagamento no Input." using (output.paymentValue > input.paymentValue)
"O valor de pagamento no Output não pode ser maior que o valor do empréstimo." using (output.paymentValue <= output.value)
```

Além disto, precisamos garantir que nenhuma outra informação foi modifica, temos que garantir que o “lender”, “borrower” e “value” continuam o mesmo.

```
"Apenas o valor de pagamento pode ser alterado." using
        (input.lender == output.lender &&
                input.borrower == output.borrower &&
                input.value == output.value)
```

No final vamos ter:

```
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>()
        when(command.value) {
            is Commands.Create -> verifyCreate(tx)
            is Commands.Pay -> verifyPay(tx)
        }
    }

    private fun verifyCreate(tx: LedgerTransaction) {
        val command = tx.commands.requireSingleCommand<Commands>()
        requireThat {
```

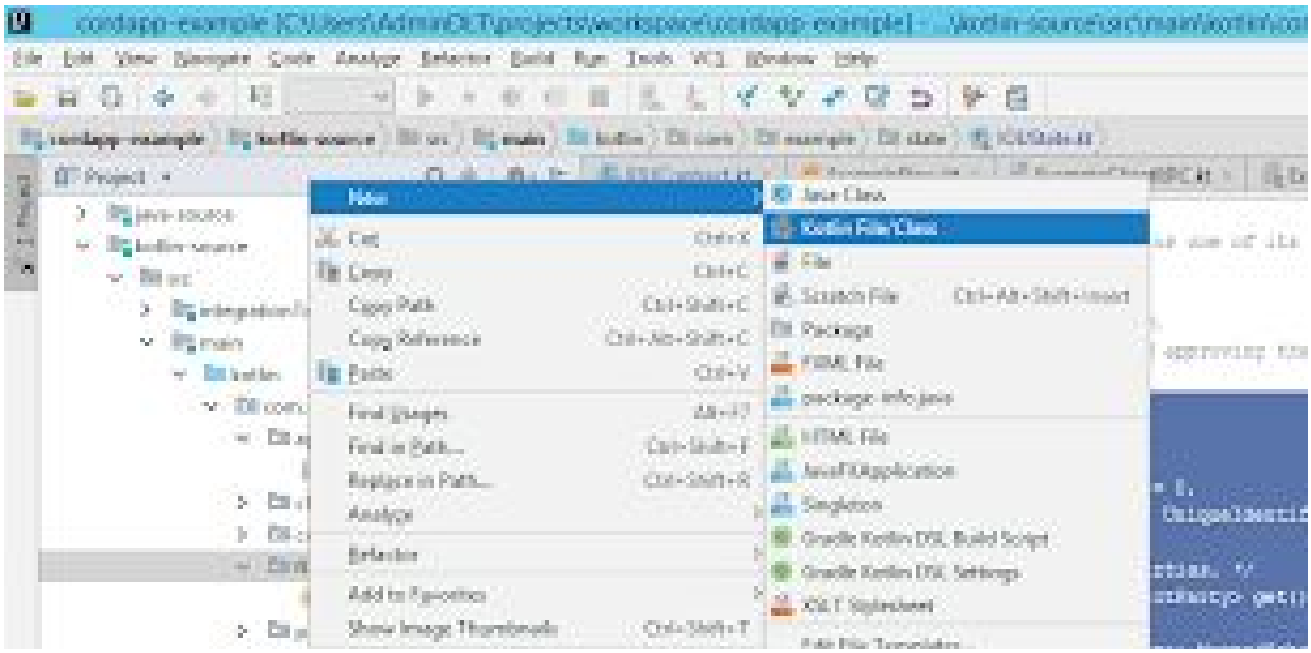
```
                // Generic constraints around the IOU transaction.
                "No inputs should be consumed when issuing an IOU." using
(tx.inputs.isEmpty())
                "Only one output state should be created." using (tx.outputs.size ==
1)
                val out = tx.outputsOfType<IOUState>().single()
                "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)
                "The IOU's payment value must be 0." using (out.paymentValue == 0)
            }
        }

        private fun verifyPay(tx: LedgerTransaction) {
            val command = tx.commands.requireSingleCommand<Commands>()
            requireThat {
                // Generic constraints around the IOU transaction.
                "Only one input should be consumed when paying an IOU." using
(tx.inputs.size == 1)
                "Only one output state should be created." using (tx.outputs.size ==
1)
                val input = tx.inputsOfType<IOUState>().single()
                val output = tx.outputsOfType<IOUState>().single()
                "The input and output state should be the same." using
(input.linearId == output.linearId)
                "All of the participants must be signers." using
(command.signers.containsAll(output.participants.map { it.owningKey }))

                // IOU-specific constraints.
                "O valor de pagamento no Output tem que ser maior que o valor de
pagamento no Input." using
                        (output.paymentValue > input.paymentValue)
                "O valor de pagamento no Output não pode ser maior que o valor do
empréstimo." using
                        (output.paymentValue <= output.value)
                "Apenas o valor de pagamento pode ser alterado." using
                        (input.lender == output.lender &&
                                input.borrower == output.borrower &&
                                input.value == output.value)
            }

        }

        /**
         * This contract only implements one command, Create.
         */
```

```
    interface Commands : CommandData {
        class Create : Commands
        class Pay: Commands
    }
}
```

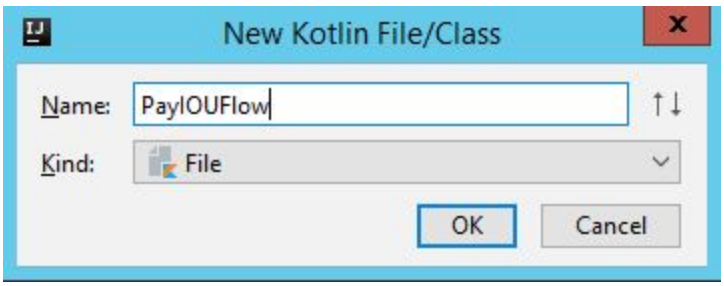
Flow

Com o *State* e o *Contrat* preparado para a receber os pagamentos, precisamos criar o fluxo de pagamento. No pacote “com.example.flow” crie um novo “Kotlin File” com o nome “PayIOUFlow”.

Pressione com o botão direito sobre o pacote e selecione as opções “New -> Kotlin File/Class”.



Digite o nome do arquivo e clique em Ok.



Vamos utilizar o modelo igual ao que está sendo utilizado no ExampleFlow, precisamos implementar um **Initiator** e um **Acceptor**.

```
package com.example.flow

import co.paralleluniverse.fibers.Suspendable
import net.corda.core.contracts.requireThat
import net.corda.core.flows.*
import net.corda.core.transactions.SignedTransaction

object PayIOUFlow {

   @InitiatingFlow
   @StartableByRPC
   class Initiator() : FlowLogic<SignedTransaction>() {

       @Suspendable
       override fun call(): SignedTransaction {
           TODO()
       }
   }

   @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 {
                   TODO()
               }
           }

           return subFlow(signTransactionFlow)
       }
   }
}
```

A primeira coisa que precisamos definir são os parâmetros que vamos receber quando este Flow for chamada. Vamos precisar de dois:

- Qual State está sendo pago
- Quanto está sendo pago

Para identificar o State, temos uma chave única que o representa, o “linearId”. O linearId é um UUIDv4, ou seja, não é possível gerar dois id’s iguais, por isso, ele é utilizado como a chave forte do contrato, então podemos pedir por parâmetro um **UUID** e um **Int** que represente o valor a ser pago.

```
class Initiator(val stateId: UUID, val paymentValue: Int)
```

Agora que temos as duas informações, precisamos buscar o State quem tem o UUID “stateId” em nosso **Vault**, precisamos primeiro definir qual a query que será feita.

No Corda para se fazer queries, é necessário construir os critérios de seleção, utilizamos então as **QueryCriterias**. Para facilitar nossa vida, conseguimos consultar LinearStates diretamente por seu UUID.

```
val criteria = QueryCriteria.LinearStateQueryCriteria(uuid = listOf(stateId))
```

Agora que temos o processo de seleção, vamos utilizar o **serviceHub** para fazer a query.

```
val oldState = serviceHub.vaultService.queryBy<IOUState>(criteria).states.single()
```

Utilizamos o método *single* para pegar apenas um State, este método vai garantir que se existir mais de 1 ou nenhum resultado, uma exceção será lançada.

Você deve ter observado que o valor que se encontra no oldState é um “StateAndRef<IOUState>”. Este tipo, armazena, tanto o seu State, quanto a Referência do seu state dentro do Chain, com informações da Transação e informação da sua altura na chain deste State.

Após selecionar o State, precisamos verificar se não estamos pagando o IOU errado, vamos verificar se realmente está é uma dívida nossa.

```
requireThat {
   "Apenas o Borrower pode pagar o IOU." using
              (ourIdentity == oldState.state.data.borrower)
}
```

A partir daqui conseguimos seguir os mesmos passos que do “ExampleFlow”, primeiro precisamos selecionar um Notary. É importante lembrar que apenas o Notary que validou o State anteriormente pode ser utilizado como Notary para as demais transações envolvendo este State, esta informação está armazenada no StateAndRef e conseguimos pegar ela facilmente.

```
val notary = oldState.state.notary
```

Agora que temos o Notary, precisamos criar o nosso novo State informando o valor de pagamento e precisamos criar o comando que vai ser utilizado para criar a transação. Vamos utilizar um recurso do Corda que nos auxilia na criação Objetos que são parecidos com objetos que já temos, a função “*copy*”, ela permite que você faça uma cópia de uma “data class” modificando apenas os valores que você deseja.

```
val newState = oldState.state.data.copy(paymentValue = paymentValue)
```

Uma outra coisa que podemos observar, é que conseguimos acessar diretamente a propriedade que queremos alterar o nome. Vamos agora criar o comando.

```
val command =
        Command(IOUContract.Commands.Pay(), newState.participants.map { it.owningKey })
```

Na criação do comando, criamos uma instância da classe que representa a nossa intenção (Pay) e precisamos informar, quais são as chaves públicas das pessoas que precisam assinar esta transação, para isso, pegamos todos os participantes das transações e acessamos o valor "*owningKey*", que é a chave pública da pessoa.

Com o comando e os dois States, precisamos agora criar a nossa Transação, para isto vamos utilizar o **TransactionBuilder**.

```
val txBuilder = TransactionBuilder(notary)
        .addInputState(oldState)
        .addOutputState(newState, oldState.state.contract)
        .addCommand(command)
```

Criamos então a transação passando nosso oldState, com as informações de referência, que serão utilizadas para validar se o State está realmente naquele estado, passando o novo State criado, e utilizando o mesmo contrato que foi utilizado no State anterior e por último passamos o Comando.

Agora já podemos verificar se não existe nenhum problema nos dados que foram criados e assinarmos a transação, antes de pedir para que os outros também assinem.

```
txBuilder.verify(serviceHub)
val partSignedTx = serviceHub.signInitialTransaction(txBuilder)
```

Podemos então coletar as assinaturas e se todos estiverem de acordo com a transação, podemos finalizá-la.

```
val flowSession = initiateFlow(newState.lender)
val fullySignedTx = subFlow(CollectSignaturesFlow(partSignedTx, setOf(flowSession)))

return subFlow(FinalityFlow(fullySignedTx))
```

Com isso finalizamos a parte do **Initiator**, agora só precisamos fazer a validação do lado do **Acceptor**.

```
@InitiatingFlow
@StartableByRPC
```

```
class Initiator(val stateId: UUID, val paymentValue: Int) :
FlowLogic<SignedTransaction>() {

   @Suspendable
   override fun call(): SignedTransaction {
       val criteria = QueryCriteria.LinearStateQueryCriteria(uuid = listOf(stateId))

       val oldState =
serviceHub.vaultService.queryBy<IOUState>(criteria).states.single()

       requireThat {
           "Apenas o Borrower pode pagar o IOU." using (ourIdentity ==
oldState.state.data.borrower)
       }

       val notary = oldState.state.notary

       val newState = oldState.state.data.copy(paymentValue = paymentValue)
       val command = Command(IOUContract.Commands.Pay(), newState.participants.map {
it.owningKey })
       val txBuilder = TransactionBuilder(notary)
               .addInputState(oldState)
               .addOutputState(newState, oldState.state.contract)
               .addCommand(command)

       txBuilder.verify(serviceHub)
       val partSignedTx = serviceHub.signInitialTransaction(txBuilder)

       val flowSession = initiateFlow(newState.lender)
       val fullySignedTx = subFlow(CollectSignaturesFlow(partSignedTx,
setOf(flowSession)))

       return subFlow(FinalityFlow(fullySignedTx))
   }
}
```

No **Acceptor**, precisamos ter certeza que a informação que foi recebida é realmente de um pagamento direcionado para a gente.

```
       "Eu devo ser o Lender deste IOU." using (output.lender == ourIdentity)
```

Checagens relacionadas ao valor do pagamento também podem ser feitas aqui, vamos exigir que o valor pago deve ser no mínimo a metade do valor total devido, garantindo assim que no máximo, o pagamento será feito em duas vezes.

```
"O pagamentos parciais precisam ser de no mínimo a metade da dívida." using
    (output.paymentValue == output.value || output.paymentValue >= (output.value / 2))
```

Também é necessário checar se já não foi feito um pagamento parcial anteriormente, para isso, vamos buscar este State em nosso vault para verificar isto.

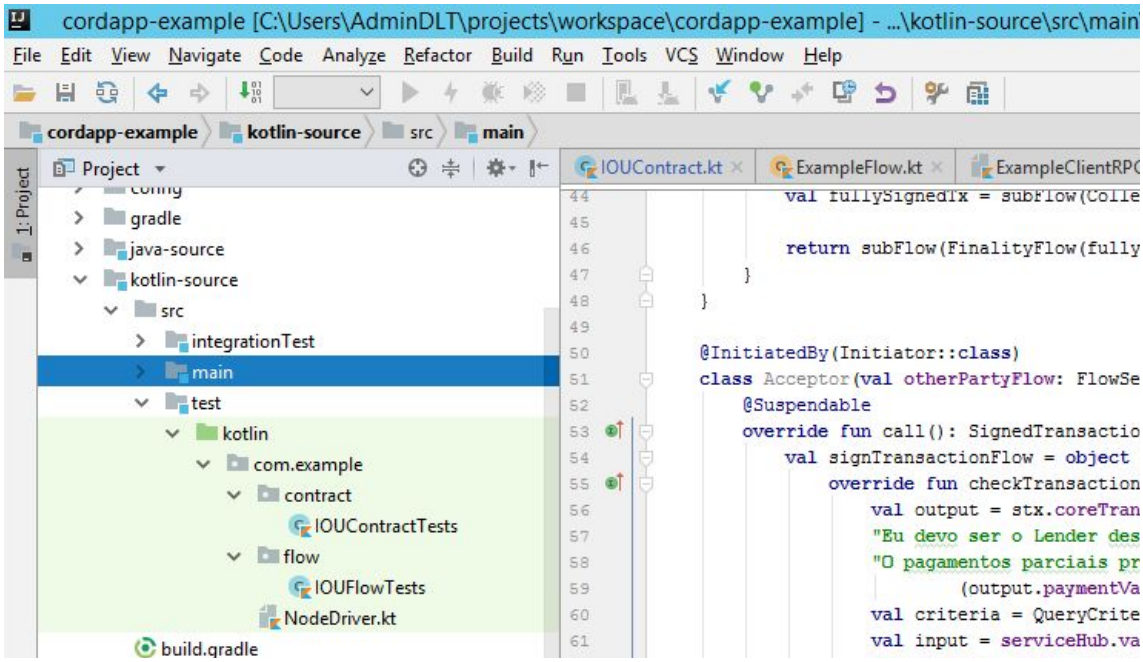
```
val criteria = QueryCriteria.LinearStateQueryCriteria(
       linearId = listOf(output.linearId))

val input =
       serviceHub.vaultService.queryBy<IOUState>(criteria).states.single().state.data

"O pagamento total deve ser realizado." using
       (output.paymentValue == output.value ||
               (output.paymentValue != output.value && input.paymentValue != 0))
```

Com mais esta validação também concluímos a parte de **Acceptor**.

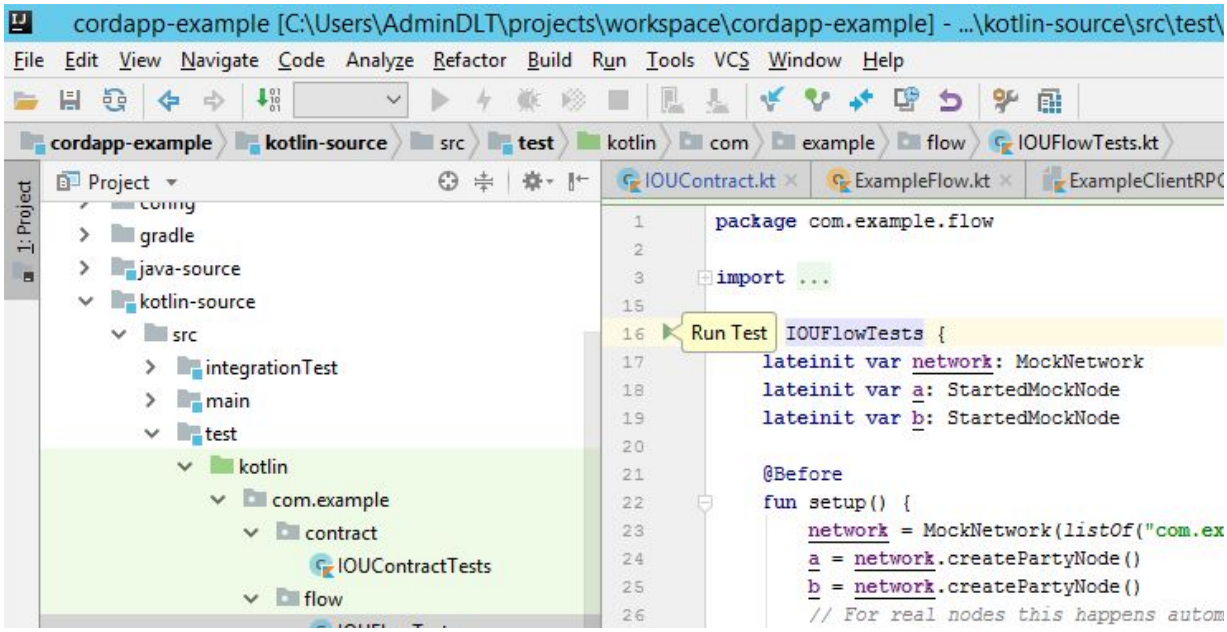
```
@InitiatedBy(Initiator::class)
class Acceptor(val otherPartyFlow: FlowSession) : FlowLogic<SignedTransaction>() {
   @Suspendable
   override fun call(): SignedTransaction {
       val signTransactionFlow = object : SignTransactionFlow(otherPartyFlow) {
           override fun checkTransaction(stx: SignedTransaction) = requireThat {
               val output = stx.coreTransaction.outputsOfType<IOUState>().single()
               "Eu devo ser o Lender deste IOU." using (output.lender == ourIdentity)
               "O pagamentos parciais precisam ser de no mínimo a metade da dívida."
using
                       (output.paymentValue == output.value || output.paymentValue >=
(output.value / 2))
               val criteria = QueryCriteria.LinearStateQueryCriteria(linearId =
listOf(output.linearId))
               val input =
serviceHub.vaultService.queryBy<IOUState>(criteria).states.single().state.data
"O pagamento total deve ser realizado." using
       (output.paymentValue == output.value ||
               (output.paymentValue != output.value && input.paymentValue != 0))

           }
       }

       return subFlow(signTransactionFlow)
   }
}
```

Testes

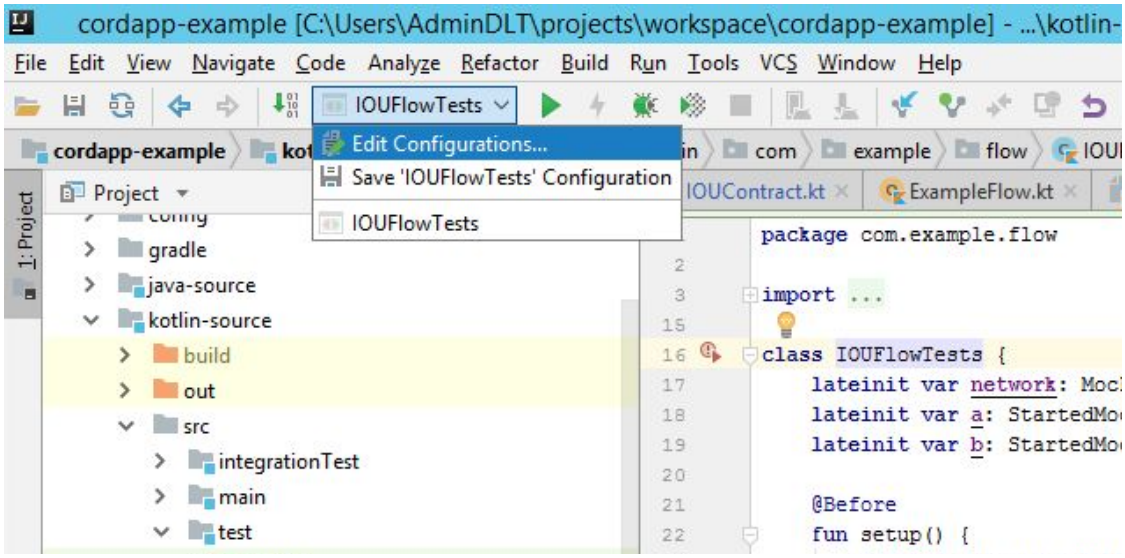
Agora que habilitamos este flow, precisamos conseguir testar para garantir que tudo está correto, fecha a pasta “main” e vamos para a pasta “test”, lá acesse “kotlin -> com.example.flow”.



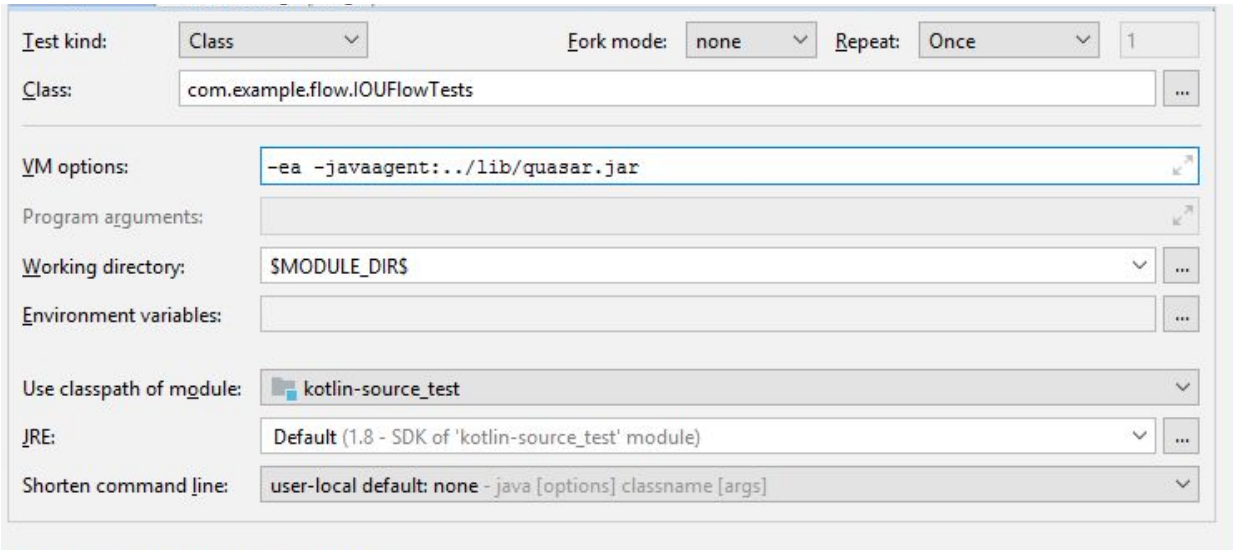
Acesse o arquivo “IOUFlowTests”. Execute os testes pressionando o botão “Run Tests”.



Caso os testes falhem, significa que precisamos alterar uma configuração de execução de testes. O Intellij, não adiciona bibliotecas externar por padrão em seus scripts de run. Precisamos adicionar então o “quasar.lib” para garantir que os testes possam ser executados.



Adicione no campo “VM options:” o valor “**-javaagent:../lib/quasar.jar**”



Execute novamente o teste para garantir que ele está passando agora.

Antes de construirmos o nosso teste, precisamos adicionar o nosso **Acceptor** no método setup.

```
listOf(a, b).forEach {
   it.registerInitiatedFlow(ExampleFlow.Acceptor::class.java)
   it.registerInitiatedFlow(PayIOUFlow.Acceptor::class.java)
}
```

Um nó normal de Corda já vai ter todos os Acceptor prontos para execução, mas como estamos trabalhando com teste, é necessário configurá-los individualmente.

Vamos agora adicionar o nosso teste, que irá garantir que o nosso Flow de pagamento está funcionando corretamente.

```
@Test
fun `deve ser possivel executar a funcao de pagamento`() {

}
```

A primeira coisa que precisamos fazer, é criar um IOU para que possamos depois fazer seu pagamento.

```
val flow = ExampleFlow.Initiator(10, b.info.singleIdentity())
val future = a.startFlow(flow)
network.runNetwork()
future.getOrThrow()
```

Agora que já temos um IOU criado, precisamos resgatar o State que foi criado, conseguimos pegar este valor do “future” que foi gerado ao inicializar o flow.

```
val stateCriado =
            future.getOrThrow().coreTransaction.outputsOfType<IOUState>().single()
```

Com este state, agora é possível chamar a função de pagamento executando o nosso flow.

```
            val payFlow = PayIOUFlow.Initiator(stateCriado.linearId.id, 10)
            val payFuture = b.startFlow(payFlow)
            network.runNetwork()
            payFuture.getOrThrow()
```

Podemos olhar agora se as informações estão presentes no Vault dos dois nós, com a informação de pagamento.

```
        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, 10)
                assertEquals(recordedState.paymentValue, 10)
                assertEquals(recordedState.lender, a.info.singleIdentity())
                assertEquals(recordedState.borrower, b.info.singleIdentity())
            }
```

```
        }
```

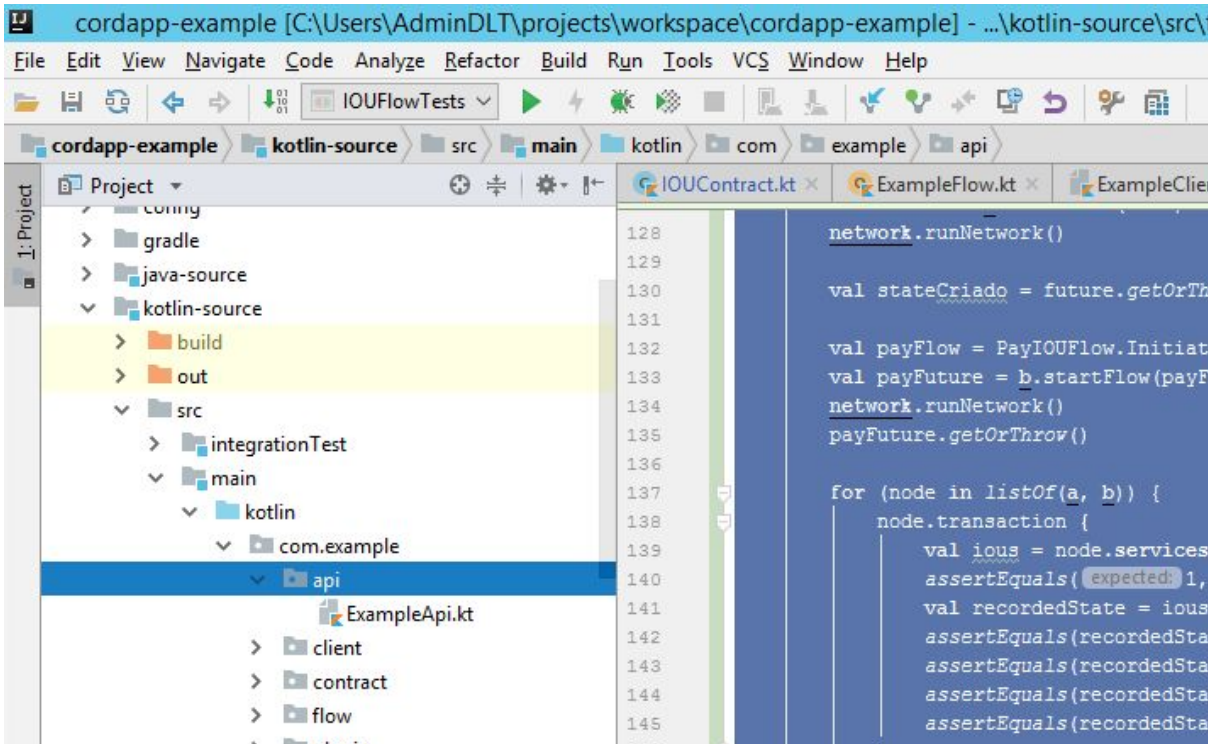
No final, o seu teste deve ficar assim:

```
@Test
fun `deve ser possivel executar a funcao de pagamento`() {
   val flow = ExampleFlow.Initiator(10, b.info.singleIdentity())
   val future = a.startFlow(flow)
   network.runNetwork()

   val stateCriado =
future.getOrThrow().coreTransaction.outputsOfType<IOUState>().single()

   val payFlow = PayIOUFlow.Initiator(stateCriado.linearId.id, 10)
   val payFuture = b.startFlow(payFlow)
   network.runNetwork()
   payFuture.getOrThrow()

   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, 10)
           assertEquals(recordedState.paymentValue, 10)
           assertEquals(recordedState.lender, a.info.singleIdentity())
           assertEquals(recordedState.borrower, b.info.singleIdentity())
       }
   }
}
```

API

Com nosso teste garantindo que a função de pagamento está funcionando, precisamos agora construir uma API para que o método possa ser acessado.

Feche a pasta “test” e volte para a pasta “main”. Lá acesse o pacote “com.example.api”.



Abra o arquivo “ExampleApi.kt”.

Neste arquivo vamos adicionar uma nova rota para que possa ser realizado o pagamento.

```
@PUT
@Path("pay-iou")
@Produces(MediaType.APPLICATION_JSON)
fun payIOU(@QueryParam("iouId") iouId: String,
           @QueryParam("payment-value")paymentValue: Int): Response {
   TODO()
}
```

Vamos receber como parâmetro o id do IOU e o valor que vai ser pago, primeiro precisamos fazer validações nos parâmetros recebidos.

```
val iouIdUUID = try {
   UUID.fromString(iouId)
} catch (e: Throwable){
   return Response.status(BAD_REQUEST).entity(
                   "Não foi possível desserializar o iouId.").build()
}
if (paymentValue <= 0)
   return Response.status(BAD_REQUEST).entity(
           "O valor de pagamento deve ser maior que zero.").build()
```

Agora, basta executar o Flow com os valores informados e retornar o ID da transação em caso de sucesso.

```
return try {
   val signedTx = rpcOps.startTrackedFlow(
       PayIOUFlow::Initiator, iouIdUUID, paymentValue).returnValue.getOrThrow()
   Response.status(CREATED).entity(
            "Transaction id ${signedTx.id} committed to ledger.\n").build()
}catch (e: Throwable){
   return Response.status(BAD_REQUEST).entity(e.message).build()
}
```

Com isso, já é possível fazer pagamentos de IOUs!