



# Agriledger – WorldBank

## CorDapp Review

Reviewed By:  
Jonathan Scialpi

Changelog:

2021-June-14	Initial report
2021-June 18	Version 2
2021-July 9	Version 3

Summary.....	3
Dashboard.....	4
Application Summary.....	4
Engagement Summary .....	4
Engagement Goals and Process.....	5
Findings Summary .....	6
1. Command Constraints.....	7
2. Notary Selection .....	8
3. Conditional Flow Constraints.....	9
4. CreateBatchFlow.....	10
5. NewSaleFlow .....	11
6. Role Based Flow Execution .....	12
7. Exception vs Flow Exception.....	14
8. Use of ENUMs for State Status .....	15
9. Test Coverage .....	17
10. Switch Statement Like Flow.....	18
11. Duplicate Items In Signatorylist .....	19
12. Responder Flows Don't Perform Validation .....	20
13. Hard Coded Participants.....	21
14. ExpectedId Param in ReceiveFinalityFlow .....	23
15. State Defined Helper Functions .....	24
16. Flow Versioning .....	25
State Transitions During Testing - CDL.....	26
1. CreateServiceRequestFlow .....	26
2. AcceptServiceRequestFlow.....	27
3. CollectorEnrouteToProducerFlow .....	28
4. CollectorWithProducerInitiatorFlow .....	29
5. CollectorEnrouteToPackhouseFlow .....	30
7. FruitFlow – Larvae Testing.....	32
8. FruitFlow – Temperature Measurement .....	33
9. FruitFlow – Grading.....	34
10. FruitFlow – Quality Inspection .....	35
11. FruitFlow – Carton Filling and Palletizing .....	36
12. FruitFlow – Shipping Details .....	37
13. FruitFlow – Shipping Details .....	38
14. CreateNewBatchFlow.....	39
Other Discussion Topics .....	40
Conclusion.....	42

# Summary

The World Bank is helping the government of Haiti, through an IDA loan, to address poverty, by helping farmers export their high quality fresh fruits, like mangoes and avocados that grow, due to optimal climate conditions, although in the most remote and poorer areas, but today are just wasted or sold locally at very low values, because they lack cold chain logistics service providers.

A blockchain/DLT solution for traceability & payments is being piloted in Haiti for the upcoming harvest season in Q1 of 2020. Agriledger, was chosen as the vendor to deliver the solution.

Agriledger will deliver a DLT/Blockchain solution to capture, trace, store and manage the end-to-end lifecycle of agricultural produce in Haiti to remove the middleman and ensure economic value is passed throughput, accounted for and received by the producer (farmer).

The proposed solution will drive the below benefits across the value chain:

1. The traceability of products/services and their production processes (for example, for purposes of food safety, product liabilities or rules of origin)
2. The reliability of a fraud safe system to keep track of all the invoicing along the value chain, until the final user, facilitating the payment and invoice financing along the supply chain.
3. Improve the Haitian producers' ability to capture the value of their products in foreign markets, without having to sell their products to intermediaries for lack of financing or technical capabilities.

Agriledger ecosystem will offer farmers opportunity to improve their quality of produce and access to more attractive foreign markets.

# Dashboard

## Application Summary

Name	Agriledger
CorDapp Version	4.3
Corda Version	5
Type	Java

## Engagement Summary

Dates	Tuesday, 1 June 2020 – Friday, 18 June 2021
Team	Professional Services - NY
Effort	1 week

# Engagement Goals and Process

R3 aims to cover the following goals as part of a CorDapp review client engagement:

- Review solution and identify areas for improvement in architecture, code design and quality assurance
- Assess the target network model to maximize scalability, performance, resiliency, and security
- Discuss client node operation processes and benchmark against R3's best practices

Cover process followed to meet these goals. This will mostly be details on the code review process and network assessment. Details that might be useful to cover:

- CDL diagrams to help understand the business workflows
- Detailed code review across Corda concepts and service as a whole

# Findings Summary

Category	Description
High	Findings which may have a material impact on the successful implementation of the Business Network or CorDapp and should be addressed as a priority
Medium	Findings which should be considered but are not likely to be showstoppers
Low	Findings which are unlikely to have a material impact on achieving your objectives

#	Title	Severity	Category
1	Command Constraints	High	Risk
2	Notary Selection	High	Risk
3	Conditional Flow Constraints	High	Risk
4	CreateBatchFlow	High	Risk
5	NewSalesFlow	High	Risk
6	Role Based Flow Execution	High	Improvement
7	Exception vs Flow Exception	Medium	Improvement
8	Use of ENUMs for State Status	Medium	Improvement
9	Test Coverage	Medium	Improvement
10	Switch Statement Like Flow	Medium	Improvement
11	Duplicated Items in SignatoryList	Low	Bug
12	Responder Flows Don't Perform Validation	Low	Improvement
13	Hard Coded Participants	Low	Improvement
14	ExpectedId Param in ReceiveFinalityFlow	Low	Improvement
15	State Defined Helper Functions	Low	Improvement
16	Flow Versioning	Low	Improvement

# 1. Command Constraints

Severity: High

Category: Risk

Target: ServiceRequestContract.kt

Description: The constraints for each of the commands can use some fortification. These constraints are the last defence between data and your ledger and every property on every state should be exhaustively checked to the point where a transaction must be perfect before every node agrees to sign.

## **Create()**

- does the provided email address fit a standard regex pattern?
- The output state status should be set to "CREATED"
- Are the LSP, Broker, etc. all comes from different owning keys IE are they not the same party

## **Accept()**

- Input state status should be "CREATED"
- Output state status should be "Assigned"
- Which values should not change from input state to output state?

## **CollectorEnRouteToProducer()**

- Input state must be "Assigned"
- Output state must be "collector enroute producer"
- enrouteToProducerData must be  $\leq$  date.now()
- enrouteAdditionalNote.length < 200

## **WithProducer()**

- Temperature between 0 and 100 degrees or some degree where the avocado would go bad?
- Ambient Temp, same as above
- Crates  $\geq$  0?
- Fruits harvested  $\geq$  0
- Fruits rejected  $\geq$  0
- advanceGiven.length < 100
- Currency in USD or BTC?
- The input state status must be "collector enroute producer"
- The output state status must be "With Producer"

## **CollectorEnRouteToPackhouse()**

- enroutePackhouseAddedOn  $\leq$  date.now()?
- enroutePackhouseAdditionalNote.length < 200
- Nothing between the input and out state has changed besides these fields and the status
- The input status must be "With Producer"
- The output status must be "Enroute Packhouse"

## 2. Notary Selection

Severity: High

Category: Risk

Target: All flows constructing a transaction builder object.

Description: The current method used to obtain a notary works in a bootstrapped network where it is assumed that only one notary will be returned in the list of available notaries. If these flows were deployed to a network containing more than one notary then the first will always be used. This does not account for a change in the order that the notaries are returned or if a notary is removed/added which can cause transactions to [unexpectedly fail](#) since the new notary would not have any of the previous transaction information, thus would not be able to prevent a double-spending attempt.

Current:

```
// We retrieve the notary identity from the network map.  
val notary = serviceHub.networkMapCache.notaryIdentities[0]
```

Recommendation:

Create a configuration file where you can define a key/value pair such as *notary = "O=Notary, L=London, C=GB"*. The configuration file is loaded on node startup and its keys/values can be obtained using the [getAppContext](#) function:

```
val notaryString = try {  
    val config: CordappConfig = services.getAppContext().config  
    config.getString("notary")  
} catch (e: CordappConfigException) {  
    throw FlowException("No notary configured, please ensure a notary is specified for the cordapp.")  
}
```



### 3. Conditional Flow Constraints

Severity: High

Category: Risk

Target: All flows which modify the *ServiceRequestState*.

Description: The Postman collection seems to have a set of flows that would result in what is commonly referred to as a “happy path” where the flows are executed in typical fashion. However, what additional logic should be considered in these flows to handle the unexpected?

The *ServiceRequestState* has several properties that can be checked within a flow such as

- temperatureReadingPackedLotData
- forcedAirCoolingEntryData
- forcedAirCoolingRemovalData
- coldRoomStorageInData
- coldRoomStorageOutData

What if one of the properties above were in breach of some contract? Would this breach have an effect on what status is used to construct the output state?

Recommendation:

Create logical checks for the values that are being passed to the flow to ensure that they are as expected for the status that is being constructed. Depending on the result, another helper function, status, or even a subflow can be used.

## 4. CreateBatchFlow

Severity: High

Category: Risk

Target: CreateBatchFlow.kt, BatchContract.kt, and ServiceRequestContract.kt,

Description: The CreateBatchFlow is responsible for producing a *BatchState*. The below screenshot highlights the use of two commands (*BatchCreation* and *CreateBatch*) with the LSP node as the required signer.



```
183 |
184 |     val serviceRequestCommand = Command(ServiceRequestContract.Commands.BatchCreation(), listOf(ourIdentity.owningKey))
185 |     val batchCommand = Command(BatchContract.BatchCommands.CreateBatch(), listOf(ourIdentity.owningKey))

26 | is BatchCommands.CreateBatch -> {
27 |     requireThat { this: Requirements
28 |         "LSP should sign the transaction" using (command.signers.count() == 1)
29 |     }
30 | }

581 | is Commands.BatchCreation -> {
582 |     requireThat { this: Requirements
583 |
584 |     }
585 | }
```

Recommendation:

- Consider combining the *BatchCreation* and *CreateBatch* commands into one since they are using the same required signers anyway. Additionally the *BatchCreation* command isn't checking anything so it doesn't seem like its required anyways.
- The *CreateBatch* command only has one constraint within it which checks for the amount of singers on the transaction. The issue here is that one constraint is not sufficient considering all the properties contained on the input/output states. Additionally, the constraint that is included does not check who signed the signature, just that only 1 participant signed it. Therefore, the transaction would pass validation if the Broker was the one who signed it instead of the LSP.
- If the *BatchCreation* command is going to stay as part of the flow, some constraints should be added which govern the way in which the *BatchState* is issued. Currently the command is blank

## 5. NewSaleFlow

Severity: High

Category: Risk

Target: NewSaleFlow.kt, BatchContract.kt, and SaleContract.kt,

Description: This flow is responsible for issuing a *SaleState* onto the ledger. There are two commands used (*SaleCreation* and *CreateSale*) each requiring the Broker to verify and sign the transaction.

```
200     val batchCommand = Command(BatchContract.BatchCommands.SaleCreation(), listOf(ourIdentity.owningKey))
201     val salesCommand = Command(SalesContract.SalesCommand.CreateSale(), listOf(ourIdentity.owningKey))
```

```
84     is BatchCommands.SaleCreation -> {
85         requireThat { this: Requirements
86             "Only Broker should sign the transaction" using (command.signers.count() == 1)
87         }
88     }
```

```
84     is BatchCommands.SaleCreation -> {
85         requireThat { this: Requirements
86             "Only Broker should sign the transaction" using (command.signers.count() == 1)
87         }
88     }
```

Recommendation:

- Consider combining the *SaleCreation* and *CreateSale* commands into one since they are using the same required signers and have the same single constraint.
- Both commands only have one constraint which checks for the amount of signers on the transaction. The issue here is that one constraint is not sufficient considering all the properties contained on the input/output states. Additionally, the constraint that is included does not check who signed the signature, just that only 1 participant signed it. Therefore, the transaction would pass validation if the LSP was the one who signed it instead of the Broker.

## 6. Role Based Flow Execution

Severity: High

Category: Improvement

Target: Flows which are expected to be ran from a specific participant such as *AcceptServiceRequestFlow* which should be ran from the LSP.

Description:

There are roles such as SAE, LSP, Collector, and Broker. Each of these roles have their own responsibility and therefore their own node/identity on the network. In the below screenshot found on lines 19-41 of *ServiceRequestState.kt* these identities are recorded to ledger and can therefore be referenced later on during flow executions:

```
@BelongsToContract(ServiceRequestContract::class)
data class ServiceRequestState(val id: String,
    val sae: Party,
    val lsp: Party,
    val collector: Party,
    val broker: Party,
    val creationData: ServiceRequestCreationModel,
    val acceptData: AcceptServiceRequestModel? = null,
    val signatories: MutableList<Party>,
    val usernames: MutableSet<String>,
    val lotRejectData: RejectLotModel? = null,
    val enrouteToProducerData: EnrouteToProducerModel? = null,
    val withProducerData: WithProducerModel? = null,
    val enrouteToPackhouseData: EnrouteToPackhouseModel? = null,
    val arrivedAtPackhouseData: ArrivedAtPackhouseModel? = null,
    val fruitFlowData: FruitFlowModel? = null,
    var batchId: String? = null,
    val proformaInvoiceData: ProformaInvoiceModel? = null,
    val saleIds: MutableList<LotSaleModel> = mutableListOf(),
    val remainingBoxes: Int? = null,
    val penaltiesIncurred: MutableList<PenaltyDataModel> = mutableListOf(),
    val firstSale: String? = null,
    override val participants: List<AbstractParty> = listOf(
        sae, lsp, collector, broker
    )
)
```

Reccomendation:

- Construct the CorDapp workflows JAR specific to the node role you are sharing it with such as including the LSP flows and sharing it only with the LSP node.

- Assuming the nodes can trust one another, the nodes' [RPC permissions](#) could be restricted to only allow the RPC users of the insurer nodes to start the flow in question.
- Add a flow constraint which checks to see if the current node running the flow (*ourIdentity*) is the same participant who is expected to run the flow. If the wrong party is found running the flow, then you can throw an exception. [Here](#) is an example. This assumes that you can trust the nodes to run the flow with this constraint.

## 7. Exception vs Flow Exception

Severity: Medium

Category: Improvement

Target: Many flows such as: *ArrivalAndDestinationFlow*, *AcceptServiceRequestFlow*, ...

Description: Throughout the flows of the CorDapp, if an issue is detected in a flow (such as a *ServiceRequestStateID* not being found in the node's vault) a *java.lang.Exception* is thrown. This class is a form of *Throwable* that indicates conditions that a reasonable application might want to catch. However, Corda flows are distributed, if an exception occurs and causes a flow to error out, the counter parties will not know of it.

Current:

```
if (existingServiceRequest != null) {  
    throw Exception("service request ${reqParam.id} already exists.")  
}
```

Recommendation:

Implement the [FlowException](#) class which will propagate the Exception to all counterparty flows and will be thrown on their end the next time they wait on a *FlowSession.receive* or *FlowSession.sendAndReceive*.

## 8. Use of ENUMs for State Status

Severity: Medium

Category: Improvement

Target: Many flows such as *CreateServiceRequest*, *AcceptServiceRequest*, ...

Description: For every status update, the newly proposed status is provided as a String parameter. This could lead to inconsistent state updates/issuances since any string is accepted.

Current:

```
val outputState = ServiceRequestState(  
    id = reqParam.id,  
    creationData = ServiceRequestCreationModel(  
        department = reqParam.department,  
        estimatedNoFruits = reqParam.estimatedNoFruits.toInt(),  
        farmerId = reqParam.farmerId,  
        producer = reqParam.producer,  
        product = Product.valueOf(reqParam.product.toUpperCase()),  
        requestedOn = LocalDateTime.parse(reqParam.requestedOn),  
        status = reqParam.status,  
        town = reqParam.town,  
        location = reqParam.location,  
        displayId = reqParam.displayId,  
        phoneNo = reqParam.phoneNo,  
        cin = reqParam.cin,  
        nif = reqParam.nif
```

Recommendation:

- [Leverage the ENUM data type](#) to define a collection of acceptable statuses for the lifecycle of a *ServiceRequestState*
- Remove the Status parameter from the payload so that only the flow logic can control the status using a helper method. Ex: *var outputState = inputState.UpdatedStatusAssigned()*
- The state status should also be verified within the Smart Contract for the given Command. For example, the *CreateServiceRequestFlow* uses the *Create()* command to issue a new *ServiceRequestState* but it is never verified that the status on the Output state is equal to "CREATED"

```
is Commands.Create -> {  
    requireThat { this: Requirements  
        "No inputs should be consumed while a Service Request is created" using (tx.inputs.isEmpty())  
        "A single Service Request should be generated as output" using (tx.outputs.size == 1)  
        "There should be a single signer" using (command.signers.count() == 1)  
        val output = tx.outputsOfType<ServiceRequestState>().single()  
        val expectedSigners = listOf(output.sae.owningKey)  
        "Service Request Id should be valid" using (!output.id.isNullOrBlank())  
        "Transaction should be signed by SAE" using (command.signers.containsAll(expectedSigners))  
    }  
}
```



## 9. Test Coverage

Severity: Medium

Category: Improvement

Target: Many flows such as *AcceptServiceRequestFlow*, *CollectorEnRouteToProducerFlow*, ...

Description:

Aside from the unit tests found in *FlowTests.kt*, most of the CorDapp's functionality does not have corresponding unit tests to ensure that they are working as expected. The Contracts and States of the CorDapp for example have no unit tests at all. All these tests combined can provide a suite of tests which safeguard your application from bugs being introduced from newly released functionality or enhancements.



```
1 package io.agriledger
2
3 import ...
4
5
6 class ContractTests {
7     private val ledgerServices = MockServices()
8
9     @Test
10    fun `dummy test`() {
11
12    }
13 }
```

Recommendation:

- Review these examples on [State Unit Tests](#) and [Contract Unit Tests](#) and apply similar concepts to this CorDapp which leverage the [MockNetwork Class](#)
- In addition to the Mock Network, consider using [Corda's Driver DSL](#) for Integration Testing where tests are run using real RPC connections with your local nodes.

## 10. Switch Statement Like Flow

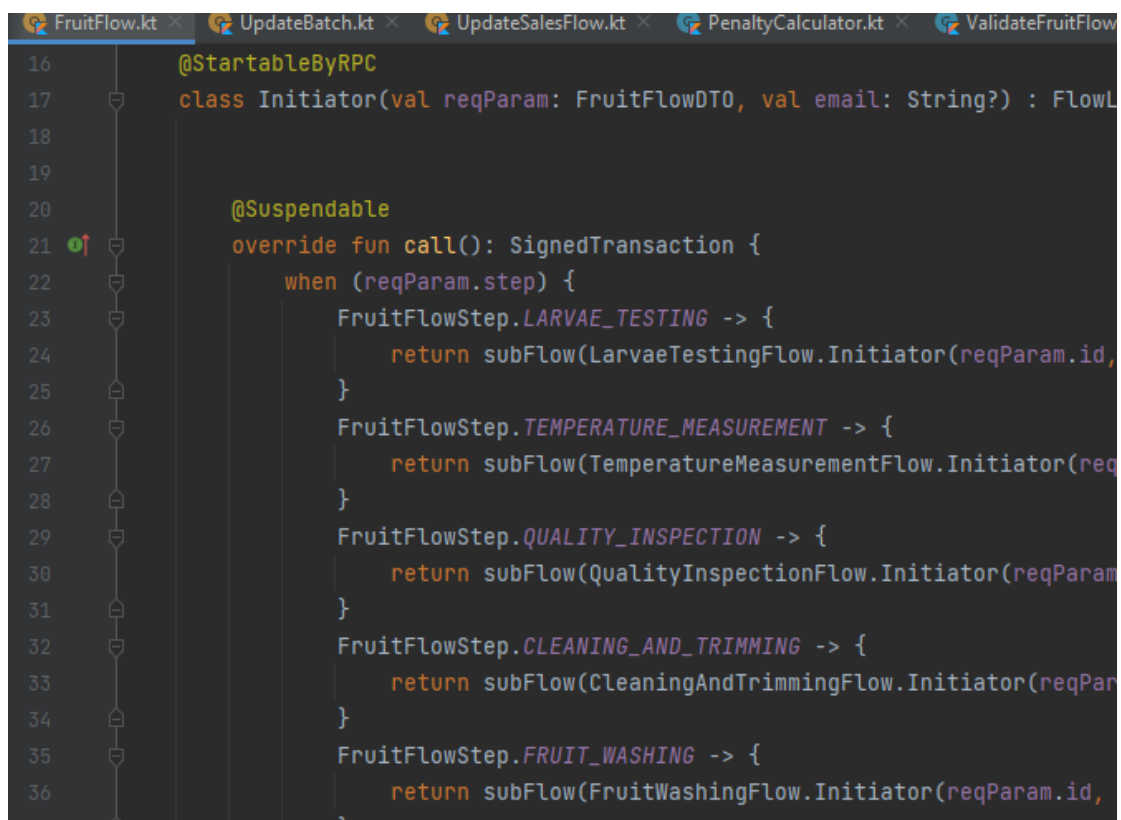
Severity: Medium

Category: Improvement

Target: Fruitflow.kt, UpdateBatch.kt, and UpdateSalesFlow.kt

Description:

These flows use a pattern that is similar to a switch statement where depending on the parameter provided (such as the *FruitFlowStep* param in *FruitFlow.kt*) another flow will be called.



```
16      @StartableByRPC
17      class Initiator(val reqParam: FruitFlowDTO, val email: String?) : FlowL
18
19
20      @Suspendable
21      override fun call(): SignedTransaction {
22          when (reqParam.step) {
23              FruitFlowStep.LARVAE_TESTING -> {
24                  return subFlow(LarvaeTestingFlow.Initiator(reqParam.id,
25              )
26              FruitFlowStep.TEMPERATURE_MEASUREMENT -> {
27                  return subFlow(TemperatureMeasurementFlow.Initiator(req
28              )
29              FruitFlowStep.QUALITY_INSPECTION -> {
30                  return subFlow(QualityInspectionFlow.Initiator(reqParam
31              )
32              FruitFlowStep.CLEANING_AND_TRIMMING -> {
33                  return subFlow(CleaningAndTrimmingFlow.Initiator(reqPar
34              )
35              FruitFlowStep.FRUIT_WASHING -> {
36                  return subFlow(FruitWashingFlow.Initiator(reqParam.id,
```

Reccomendation:

It is the Spring controller's job to intercept incoming requests and send the data to the appropriate model for processing. However, the way these flows are designed is like that of a controller and this functionality can be exported to the controller instead which will call one less flow per request. These additional flow requests could hinder performance as this flow would have to be checkpointed/suspended before calling the subflows. Additionally, there could be issues regarding suspended flows if any of the subflow were to exit abnormally.

## 11. Duplicate Items In Signatorylist

Severity: Low

Category: Bug

Target: Many flows such as *AcceptServiceRequestFlow*, *CollectorEnRouteToProducerFlow*, ...

Description: Each time a flow is ran the current node's identity is added to the *SignatoryList* property on the *ServiceRequestState*. Since some flows are ran by the same node, this leads to the signatory list growing to unnecessary size due to duplicate members in the list. Here is the signatory list retrieved from a vault query after running through all the flows:

signatories:

- "O=SAE, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=Collector, L=Port-au-Prince, C=HT"
- "O=Collector, L=Port-au-Prince, C=HT"
- "O=Collector, L=Port-au-Prince, C=HT"
- "O=Collector, L=Port-au-Prince, C=HT"
- "O=Collector, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"
- "O=LSP, L=Port-au-Prince, C=HT"

Current:

```
val newSignatories = vaultData.signatories.toMutableList()
newSignatories.add(ourIdentity)
newSignatories.add(collector)
```

Recommendation:

- Use a [HashSet](#) to ensure that the Signatory list never contains duplicates.
- If this functionality was by design, it might not be necessary as the required signers of every transaction are retrievable from a [SignedTransaction Object](#).

`requiredSigningKeys`

`val requiredSigningKeys: Set<PublicKey>`  
Specifies all the public keys that require signatures for the transaction to be valid.

`sigs`

`val sigs: List<TransactionSignature>`  
a list of signatures from individual (non-composite) public keys. This is passed as a list of signatures when verifying composite key signatures, but may be used as individual signatures where a single key is expected to sign.

## 12. Responder Flows Don't Perform Validation

Severity: Low

Category: Improvement

Target: Many flows such as *AcceptServiceRequestFlow*, *TransportDetailsFlow*, ...

Description:

Most responder flows in seem to be very light on transaction verification before signing.

```
@InitiatedBy(Initiator::class)
class Responder(val counterpartySession: FlowSession) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        // Responder flow logic goes here.
        subFlow(ReceiveFinalityFlow(counterpartySession))
    }
}
```

Recommendation:

Consider implementing some basic checks such as ensuring that there is one output state and that output state is of type *ServiceRequestState*.

## 13. Hard Coded Participants

Severity: Low

Category: Improvement

Target: Many flows such as *CreateServiceRequest*, *AcceptedServiceRequest*, ...

Description: The identity of each participant besides the current node running the flow is obtained via constant variable defined in a file called *FlowConstants.kt*.

Transactions will succeed as its currently implemented however if there is any need to update the participants or their properties then a new CorDapp version will be needed.

The *CreateServiceRequestFlow.kt* is probably the only flow that can make a case for using the below implementation since the *ServiceRequestState* does not exist yet. However, once it is created, other flows which reference this state can pull the Party objects off the input state.

Current:

```
val lsp: Party = serviceHub.identityService.wellKnownPartyFromX500Name(FlowConstants.lspName)
?: throw IllegalArgumentException("Couldn't find counterparty for LSP in identity service")

val collector: Party = serviceHub.identityService.wellKnownPartyFromX500Name(FlowConstants.collectorName)
?: throw IllegalArgumentException("Couldn't find counterparty for Collector in identity service")

val broker: Party = serviceHub.identityService.wellKnownPartyFromX500Name(FlowConstants.brokerName)
?: throw IllegalArgumentException("Couldn't find counterparty for Broker in identity service")
```

Recommendation:

- Consider taking these participants as parameters which the front end can pass as part of the payload in the RPC.
- If its preferred not include them as parameters, then these values can be set in the configuration file recommended for the notary as key/value pairs. Changes to these values would still require a node restart since configuration files are loaded on startup but at least you would not have to modify and redistribute your code.

The above two bullets are just for issuing the *ServiceRequestState*. Once this state is created, the input state's properties can be referenced for participants that already exist there such as LSP, Collector, and Broker.

```
data class ServiceRequestState(val id: String,  
                                val sae: Party,  
                                val lsp: Party,  
                                val collector: Party,  
                                val broker: Party,
```

## 14. ExpectedId Param in ReceiveFinalityFlow

Severity: Low

Category: Improvement

Target: CreateServiceRequestFlow – line 134

Description:

Although this is not a required parameter, supplying the expected SignedTransactionID can give responders further confidence that the transaction was processed as expected since the output transaction ID matches the same that the responder calculated.

Current:

```
129         @InitiatedBy(Initiator::class)
130         class Responder(val counterpartySession: FlowSession) : FlowLogic<Unit>() {
131             @Suspendable
132             override fun call() {
133                 // Responder flow logic goes here.
134                 subFlow(ReceiveFinalityFlow(counterpartySession))
135             }
136         }
137     }
```

Recommendation:

Refer to the [below example](#) on how to use the expectedId parameter.

Example - checking and signing a transaction involving a [net.corda.core.contracts.DummyContract](#), see [CollectSignaturesFlowTests.kt](#) for further exam

```
class Responder(val otherPartySession: FlowSession): FlowLogic<SignedTransaction>() {
    @Suspendable override fun call(): SignedTransaction {
        // [SignTransactionFlow] sub-classed as a singleton object.
        val flow = object : SignTransactionFlow(otherPartySession) {
            @Suspendable override fun checkTransaction(stx: SignedTransaction) = requireThat {
                val tx = stx.tx
                val magicNumberState = tx.outputs.single().data as DummyContract.MultiOwnerState
                "Must be 1337 or greater" using (magicNumberState.magicNumber >= 1337)
            }
        }
        // Invoke the subFlow, in response to the counterparty calling [CollectSignaturesFlow].
        val expectedTxId = subFlow(flow).id
        return subFlow(ReceiveFinalityFlow(otherPartySession, expectedTxId))
    }
}
```

## 15. State Defined Helper Functions

Severity: Low

Category: Improvement

Target: State Objects such as *ServiceRequestState*.

Description: All state updates are done adhoc using the *copy()* function within flows. This does not ensure consistent functionality as flow developers will create their own method to update the state and deploy. To improve consistency, helper functions can be created with a specific intent. If we look at the *AcceptServiceRequest* flow for example, we see that a *reqParam* object is passed to the flow to create a *ServiceRequestState* output state on line 95. Some of *reqParam*'s properties are used to create a copy.

This copy function could be wrapped by another function called *AcceptServiceRequest* which accepts the necessary parameters to update the state and this function can be defined on the *ServiceRequestState* object itself so that all nodes with access to the contracts jar file of the *CorDapp* can update the state the same way any other node participating in the *CorDapp* would.

Current:

```
val outputState: ServiceRequestState = vaultData.copy(  
    acceptData = AcceptServiceRequestModel(  
        pickupAddedOn = LocalDateTime.parse(reqParam.pickupAddedOn),  
        collectorName = reqParam.collectorName,  
        collectionPoint = reqParam.collectionPoint,  
        dropOffAtPackhouse = reqParam.dropOffAtPackhouse,  
        dateOfPickup = LocalDateTime.parse(reqParam.dateOfPickup),  
        scheduled = reqParam.scheduled,  
        amount = reqParam.amount.toInt(),  
        scheduledAfter = reqParam.scheduledAfter.toInt()  
    ),  
    creationData = vaultData.creationData.copy(status = reqParam.status),  
    signatories = newSignatories,  
    usernames = usernames
```

Recommendation:

- The recommendation is to create [wrapper functions](#) which can help ensure that all nodes participating on transactions can create state proposals in a consistent manor and with less repeated code.



## 16. Flow Versioning

Severity: Low

Category: Improvement

Target: All flows

Description: Apply [versions to flows](#) to signal changes that are not backwards compatible.

Recommendation:

### Flow versioning ⇔

Any flow that initiates other flows must be annotated with the `@InitiatingFlow` annotation, which is defined as:

```
annotation class InitiatingFlow(val version: Int = 1)
```

The `version` property, which defaults to 1, specifies the flow's version. This integer value should be incremented whenever there is a release of a flow which has changes that are not backwards-compatible. A non-backwards compatible change is one that changes the interface of the flow.

# State Transitions During Testing - CDL

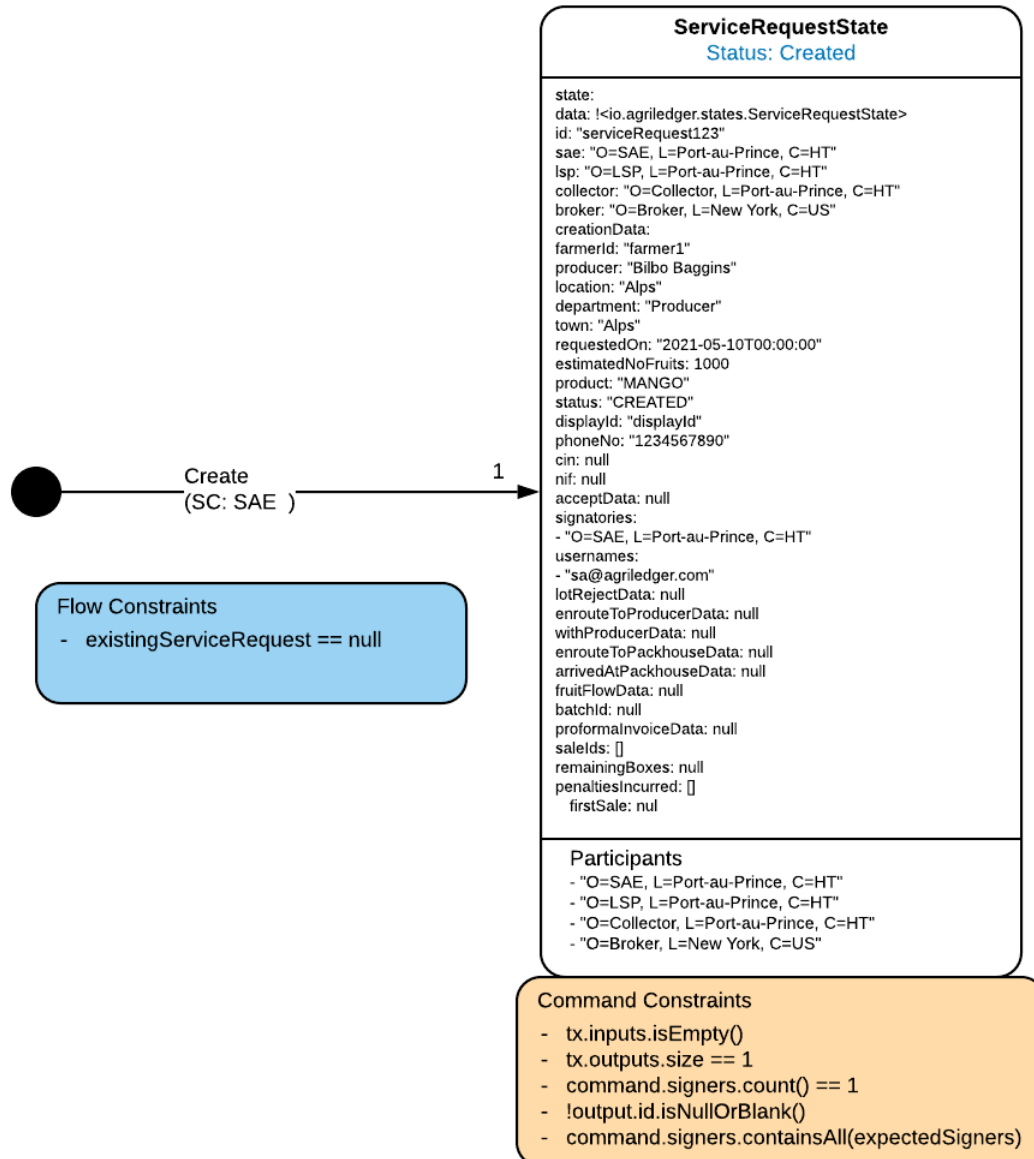
## 1. CreateServiceRequestFlow

Node: SAE

Endpoint: POST - create-servicerequest

Spring Host: 50005

RPC Port: 10012



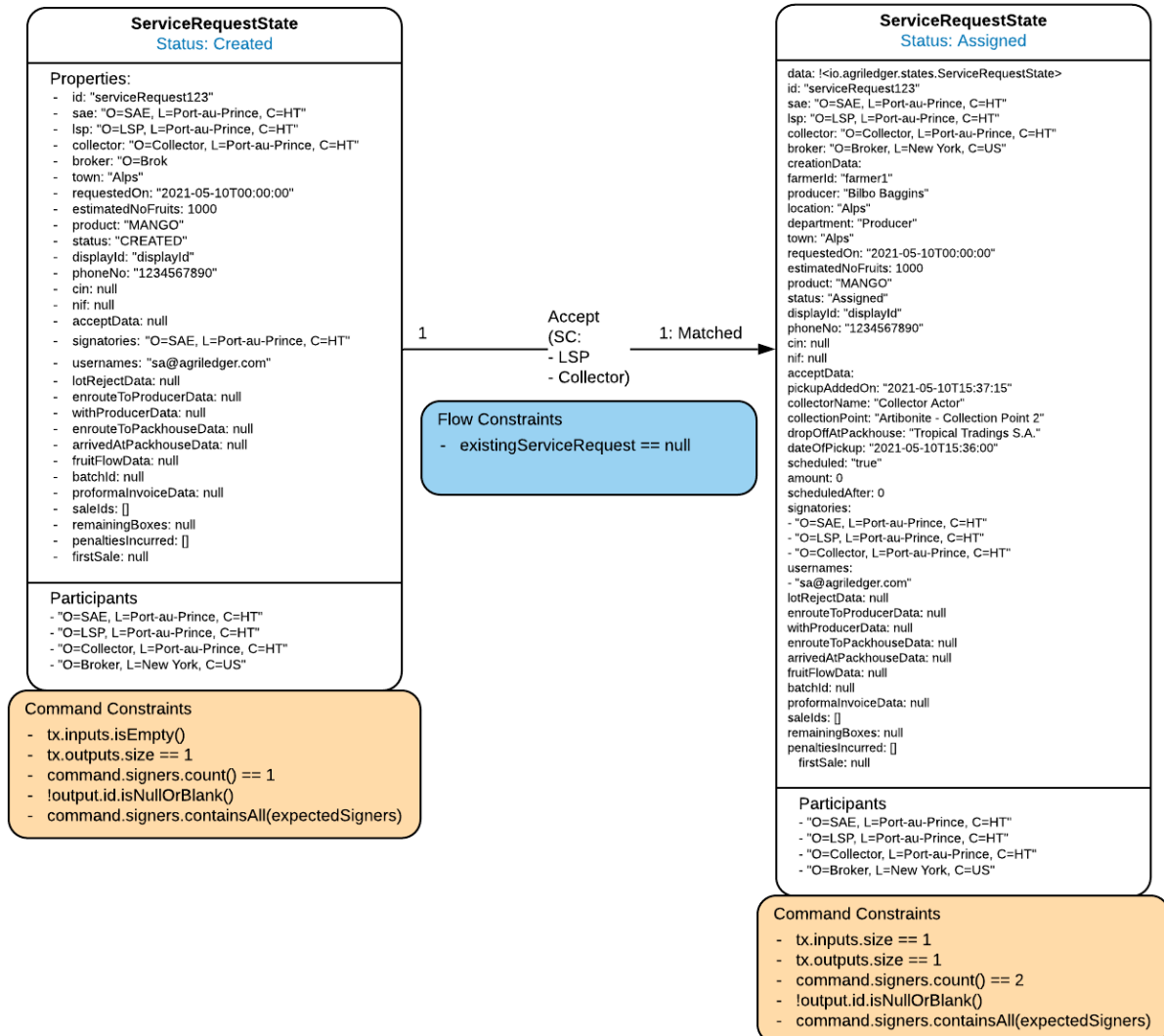
## 2. AcceptServiceRequestFlow

Node: LSP

Endpoint: POST - accept-service-request

Spring Host: 50007

RPC Port: 10006



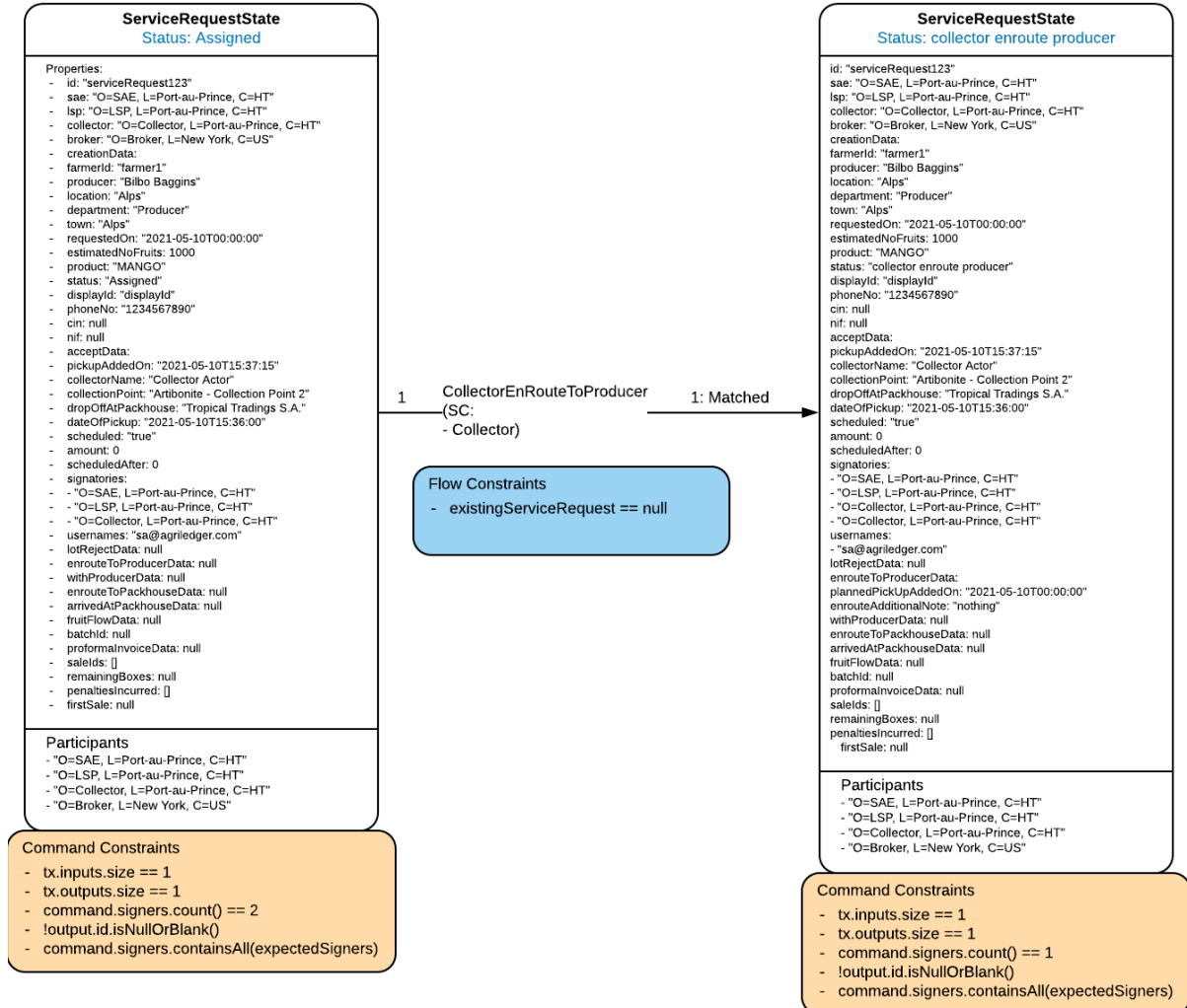
### 3. CollectorEnrouteToProducerFlow

Node: Collector

Endpoint: POST - collector-en-route-to-producer

Spring Host: 50009

RPC Port: 10011



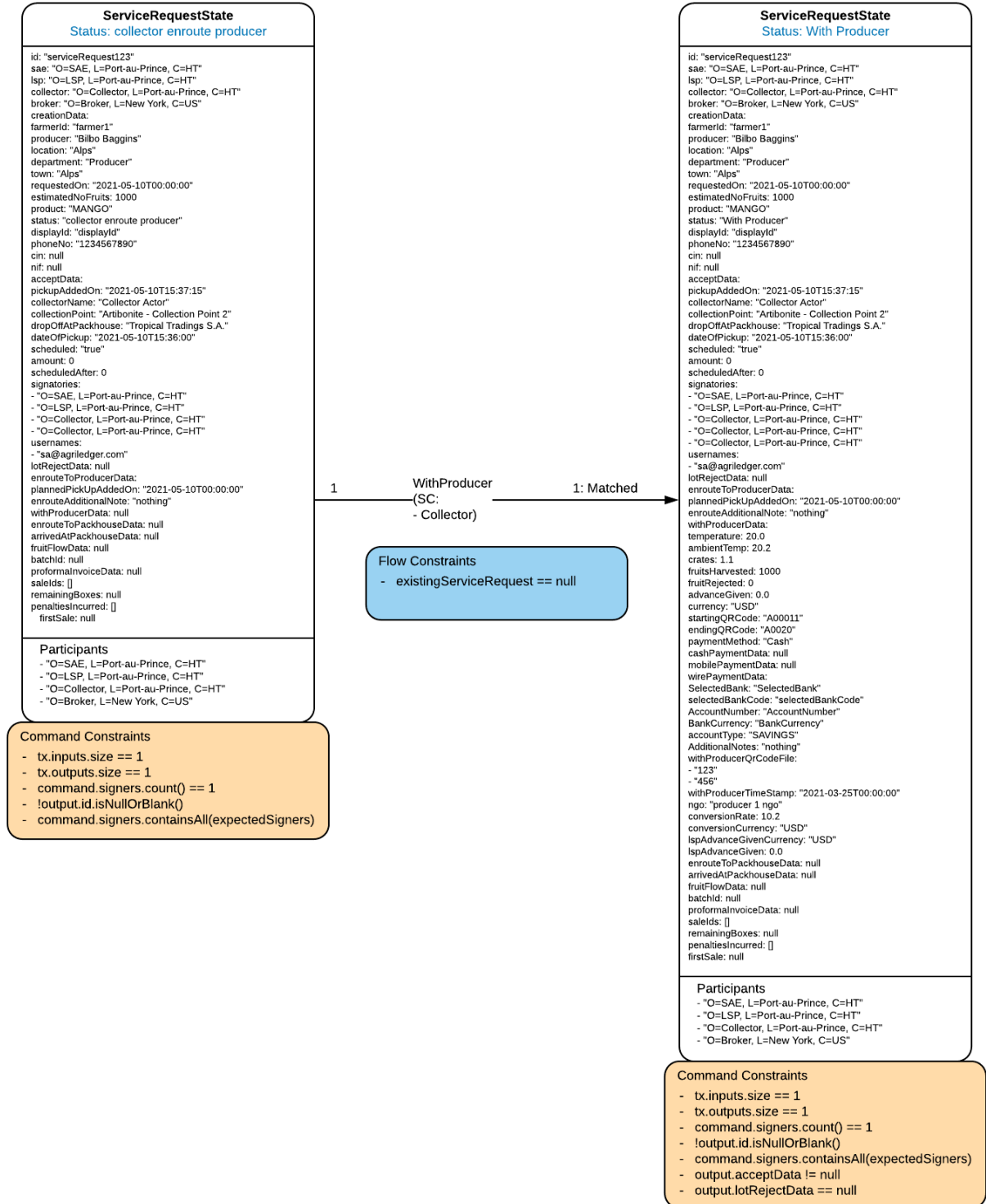
## 4. CollectorWithProducerInitiatorFlow

Node: Collector

Endpoint: POST - with-producer

Spring Host: 50009

RPC Port: 10011



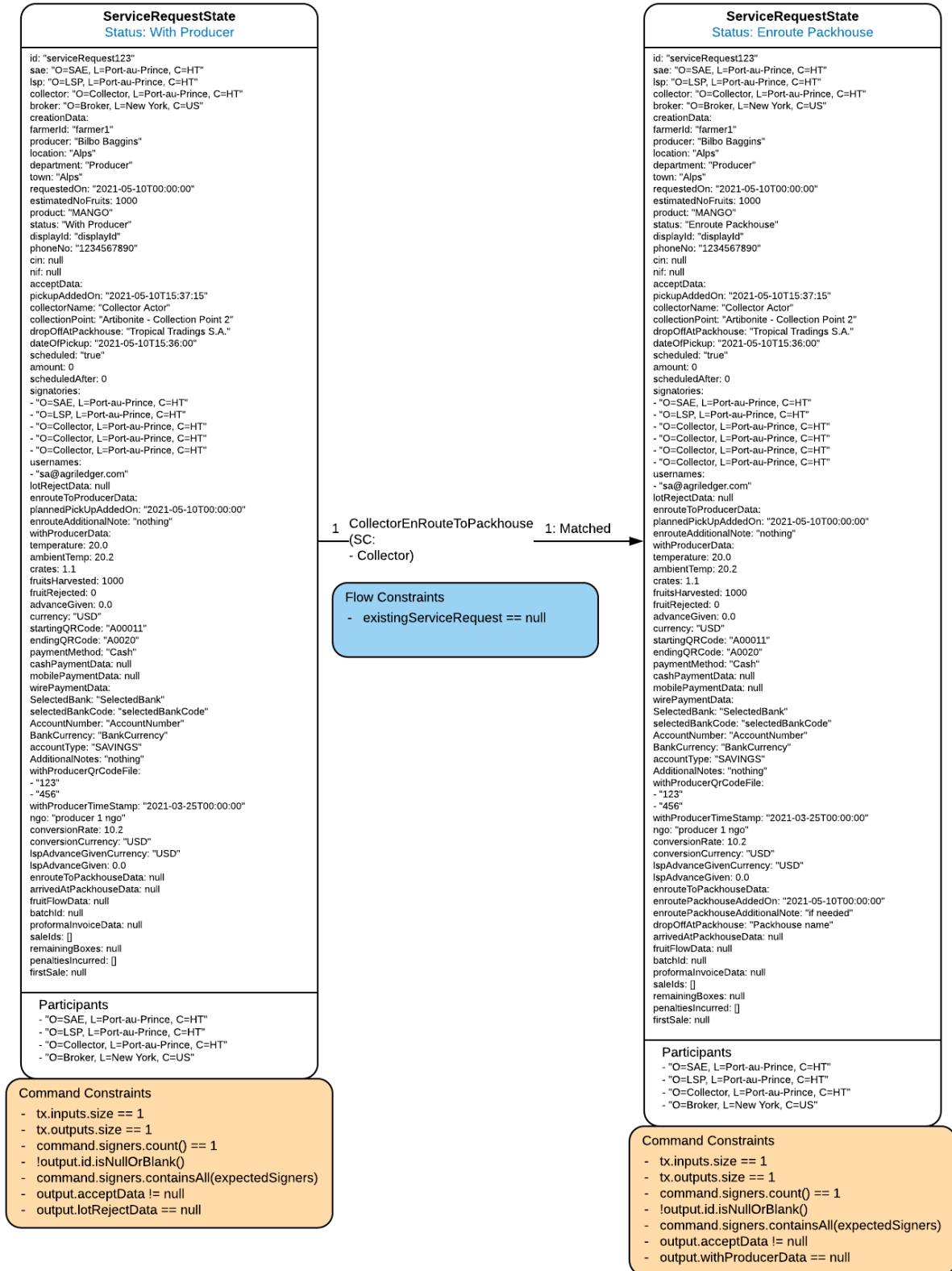
## 5. CollectorEnrouteToPackhouseFlow

Node: Collector

Endpoint: POST - collector-enroute-packhouse

Spring Host: 50009

RPC Port: 10011



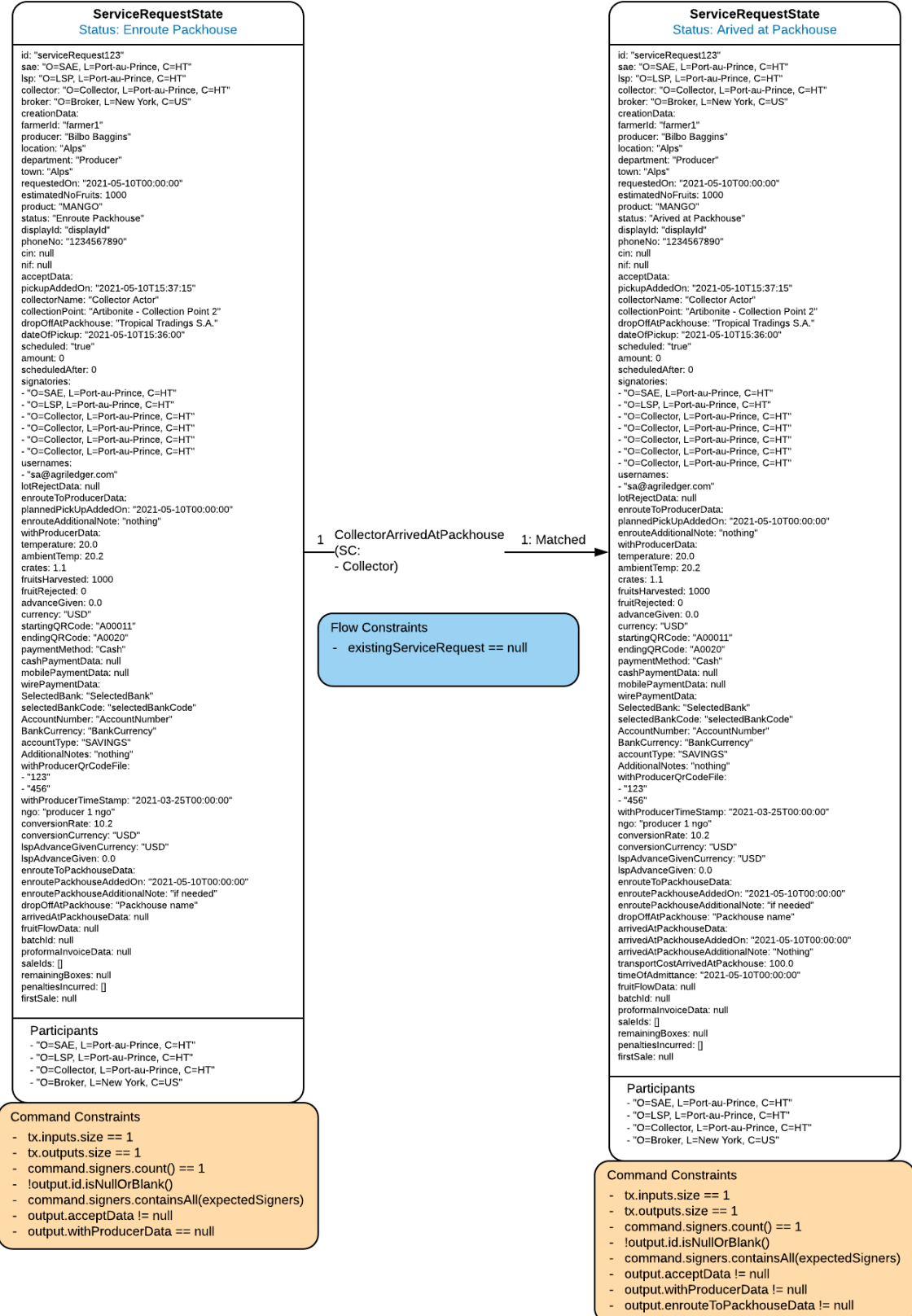
## 6. CollectorArrivedAtPackHouseFlow

Node: Collector

Endpoint: POST - collector-arrived-at-packhouse

Spring Host: 50009

RPC Port: 10011





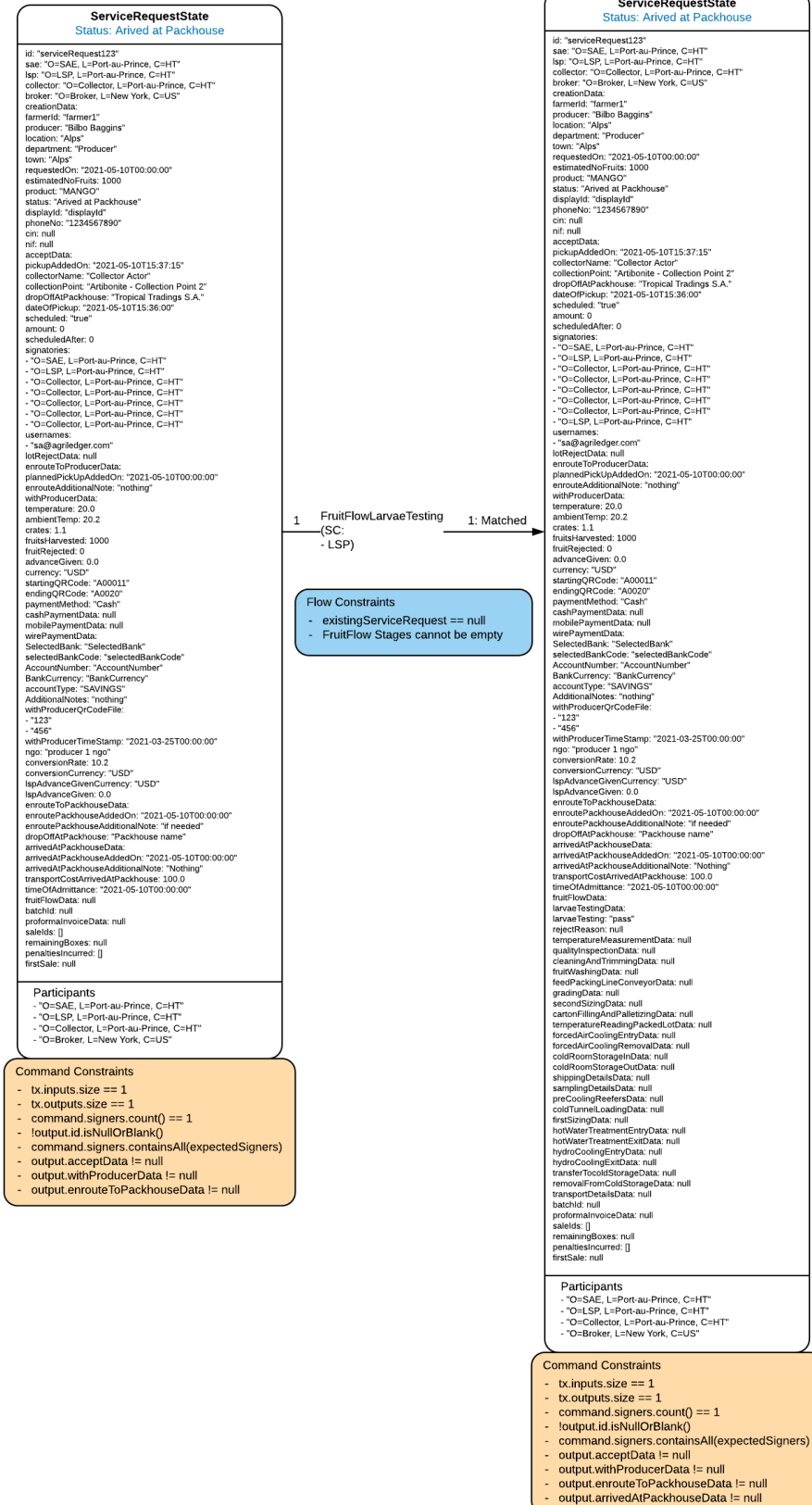
## 7. FruitFlow – Larvae Testing

Node: LSP

Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006





## 8. FruitFlow – Temperature Measurement

Node: LSP

Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006

ServiceRequestState  
Status: Arrived at Packhouse

id: "serviceRequest123"  
sae: "O=SAE, L=Port-au-Prince, C=HT"  
lsp: "O=LSP, L=Port-au-Prince, C=HT"  
collector: "O=Collector, L=Port-au-Prince, C=HT"  
broker: "O=Broker, L=New York, C=US"  
creationData:  
farmerId: "farmer1"  
producer: "Bilbo Baggins"  
location: "Alps"  
department: "Producer"  
town: "Alps"  
requestedOn: "2021-05-10T00:00:00"  
estimatedNoFruits: 1000  
product: "MANGO"  
status: "Arrived at Packhouse"  
displayId: "displayId"  
phoneNo: "1234567890"  
crt: null  
nft: null  
acceptData:  
pickupAddedOn: "2021-05-10T15:37:15"  
collectorName: "Collector Actor"  
collectionPoint: "Arbitonite - Collection Point 2"  
dropOffAtPackhouse: "Tropical Tradings S.A."  
dateOfPickup: "2021-05-10T15:36:00"  
scheduled: "true"  
amount: 0  
scheduledAfter: 0  
signatories:  
- "O=SAE, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
usernames:  
- "sa@agiledger.com"  
lotRejectData: null  
enrouteToProducerData:  
plannedPickUpAddedOn: "2021-05-10T00:00:00"  
enrouteAdditionalNote: "nothing"  
withProducerData:  
temperature: 20.0  
ambientTemp: 20.2  
crates: 1.1  
fruitsHarvested: 1000  
fruitRejected: 0  
advanceGiven: 0.0  
currency: "USD"  
startingQRCode: "A00011"  
endingQRCode: "A00020"  
paymentMethod: "Cash"  
cashPaymentData: null  
mobilePaymentData: null  
wirePaymentData:  
SelectedBank: "SelectedBank"  
selectedBankCode: "selectedBankCode"  
AccountNumber: "AccountNumber"  
BankCurrency: "BankCurrency"  
accountType: "SAVINGS"  
AdditionalNotes: "nothing"  
withProducerQRCodeFile:  
- "123"  
- "456"  
withProducerTimeStamp: "2021-03-25T00:00:00"  
ngo: "producer 1 ngo"  
conversionRate: 10.2  
conversionCurrency: "USD"  
lspAdvanceGivenCurrency: "USD"  
lspAdvanceGiven: 0.0  
enrouteToPackhouseData:  
enroutePackhouseAddedOn: "2021-05-10T00:00:00"  
enroutePackhouseAdditionalNote: "If needed"  
dropOffAtPackhouse: "Packhouse name"  
arrivedAtPackhouseData:  
arrivedAtPackhouseAddedOn: "2021-05-10T00:00:00"  
arrivedAtPackhouseAdditionalNote: "Nothing"  
transportCostArrivedAtPackhouse: 100.0  
timeOfAdmittance: "2021-05-10T00:00:00"  
fruitFlowData:  
larvaeTestingData:  
larvaeTesting: "pass"  
rejectReason: null  
temperatureMeasurementData: null  
qualityInspectionData: null  
cleaningAndTrimmingData: null  
fruitWashingData: null  
feedPackingLineConveyorData: null  
gradingData: null  
secondSizingData: null  
cartonFillingAndPalletizingData: null  
temperatureReadingPackedLotData: null  
forcedAirCoolingEntryData: null  
forcedAirCoolingRemovalData: null  
coldRoomStorageInData: null  
coldRoomStorageOutData: null  
shippingDetailsData: null  
samplingDetailsData: null  
preCoolingReefersData: null  
coldTunnelLoadingData: null  
firstSizingData: null  
hotWaterTreatmentEntryData: null  
hotWaterTreatmentExitData: null  
hydroCoolingEntryData: null  
transferToColdStorageData: null  
removalFromColdStorageData: null  
transportDetailsData: null  
batchId: null  
proformaInvoiceData: null  
saleIds: []  
remainingBoxes: null  
penaltiesIncurred: []  
firstSale: null

Participants  
- "O=SAE, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Broker, L=New York, C=US"

Command Constraints  
- tx.inputs.size == 1  
- tx.outputs.size == 1  
- command.signers.count() == 1  
- !output.id.isNullOrBlank()  
- command.signers.containsAll(expectedSigners)  
- output.acceptData != null  
- output.withProducerData != null  
- output.enrouteToPackhouseData != null  
- output.arrivedAtPackhouseData != null

1  
FruitFlowTemperatureMeasurement  
(SC:  
- LSP)

Flow Constraints  
- existingServiceRequest == null  
- FruitFlow Stages cannot be empty

1: Matched

ServiceRequestState  
Status: Arrived at Packhouse

id: "serviceRequest123"  
sae: "O=SAE, L=Port-au-Prince, C=HT"  
lsp: "O=LSP, L=Port-au-Prince, C=HT"  
collector: "O=Collector, L=Port-au-Prince, C=HT"  
broker: "O=Broker, L=New York, C=US"  
creationData:  
farmerId: "farmer1"  
producer: "Bilbo Baggins"  
location: "Alps"  
department: "Producer"  
town: "Alps"  
requestedOn: "2021-05-10T00:00:00"  
estimatedNoFruits: 1000  
product: "MANGO"  
status: "Arrived at Packhouse"  
displayId: "displayId"  
phoneNo: "1234567890"  
crt: null  
nft: null  
acceptData:  
pickupAddedOn: "2021-05-10T15:37:15"  
collectorName: "Collector Actor"  
collectionPoint: "Arbitonite - Collection Point 2"  
dropOffAtPackhouse: "Tropical Tradings S.A."  
dateOfPickup: "2021-05-10T15:36:00"  
scheduled: "true"  
amount: 0  
scheduledAfter: 0  
signatories:  
- "O=SAE, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
usernames:  
- "sa@agiledger.com"  
lotRejectData: null  
enrouteToProducerData:  
plannedPickUpAddedOn: "2021-05-10T00:00:00"  
enrouteAdditionalNote: "nothing"  
withProducerData:  
temperature: 20.0  
ambientTemp: 20.2  
crates: 1.1  
fruitsHarvested: 1000  
fruitRejected: 0  
advanceGiven: 0.0  
currency: "USD"  
startingQRCode: "A00011"  
endingQRCode: "A00020"  
paymentMethod: "Cash"  
cashPaymentData: null  
mobilePaymentData: null  
wirePaymentData:  
SelectedBank: "SelectedBank"  
selectedBankCode: "selectedBankCode"  
AccountNumber: "AccountNumber"  
BankCurrency: "BankCurrency"  
accountType: "SAVINGS"  
AdditionalNotes: "nothing"  
withProducerQRCodeFile:  
- "123"  
- "456"  
withProducerTimeStamp: "2021-03-25T00:00:00"  
ngo: "producer 1 ngo"  
conversionRate: 10.2  
conversionCurrency: "USD"  
lspAdvanceGivenCurrency: "USD"  
lspAdvanceGiven: 0.0  
enrouteToPackhouseData:  
enroutePackhouseAddedOn: "2021-05-10T00:00:00"  
enroutePackhouseAdditionalNote: "If needed"  
dropOffAtPackhouse: "Packhouse name"  
arrivedAtPackhouseData:  
arrivedAtPackhouseAddedOn: "2021-05-10T00:00:00"  
arrivedAtPackhouseAdditionalNote: "Nothing"  
transportCostArrivedAtPackhouse: 100.0  
timeOfAdmittance: "2021-05-10T00:00:00"  
fruitFlowData:  
larvaeTestingData:  
larvaeTesting: "pass"  
rejectReason: null  
temperatureMeasurementData:  
ambientTemp: 22.0  
internalFruitTemp: 22.0  
isTemperatureBreach: false  
temperatureBreachCount: 0  
qualityInspectionData: null  
cleaningAndTrimmingData: null  
fruitWashingData: null  
feedPackingLineConveyorData: null  
gradingData: null  
secondSizingData: null  
cartonFillingAndPalletizingData: null  
temperatureReadingPackedLotData: null  
forcedAirCoolingEntryData: null  
forcedAirCoolingRemovalData: null  
coldRoomStorageInData: null  
coldRoomStorageOutData: null  
shippingDetailsData: null  
samplingDetailsData: null  
preCoolingReefersData: null  
coldTunnelLoadingData: null  
firstSizingData: null  
hotWaterTreatmentEntryData: null  
hotWaterTreatmentExitData: null  
hydroCoolingEntryData: null  
hydroCoolingExitData: null  
transferToColdStorageData: null  
removalFromColdStorageData: null  
transportDetailsData: null  
batchId: null  
proformaInvoiceData: null  
saleIds: []  
remainingBoxes: null  
penaltiesIncurred: []  
firstSale: null

Participants  
- "O=SAE, L=Port-au-Prince, C=HT"  
- "O=LSP, L=Port-au-Prince, C=HT"  
- "O=Collector, L=Port-au-Prince, C=HT"  
- "O=Broker, L=New York, C=US"

Command Constraints  
- tx.inputs.size == 1  
- tx.outputs.size == 1  
- command.signers.count() == 1  
- !output.id.isNullOrBlank()  
- command.signers.containsAll(expectedSigners)  
- output.acceptData != null  
- output.withProducerData != null  
- output.enrouteToPackhouseData != null  
- output.arrivedAtPackhouseData != null

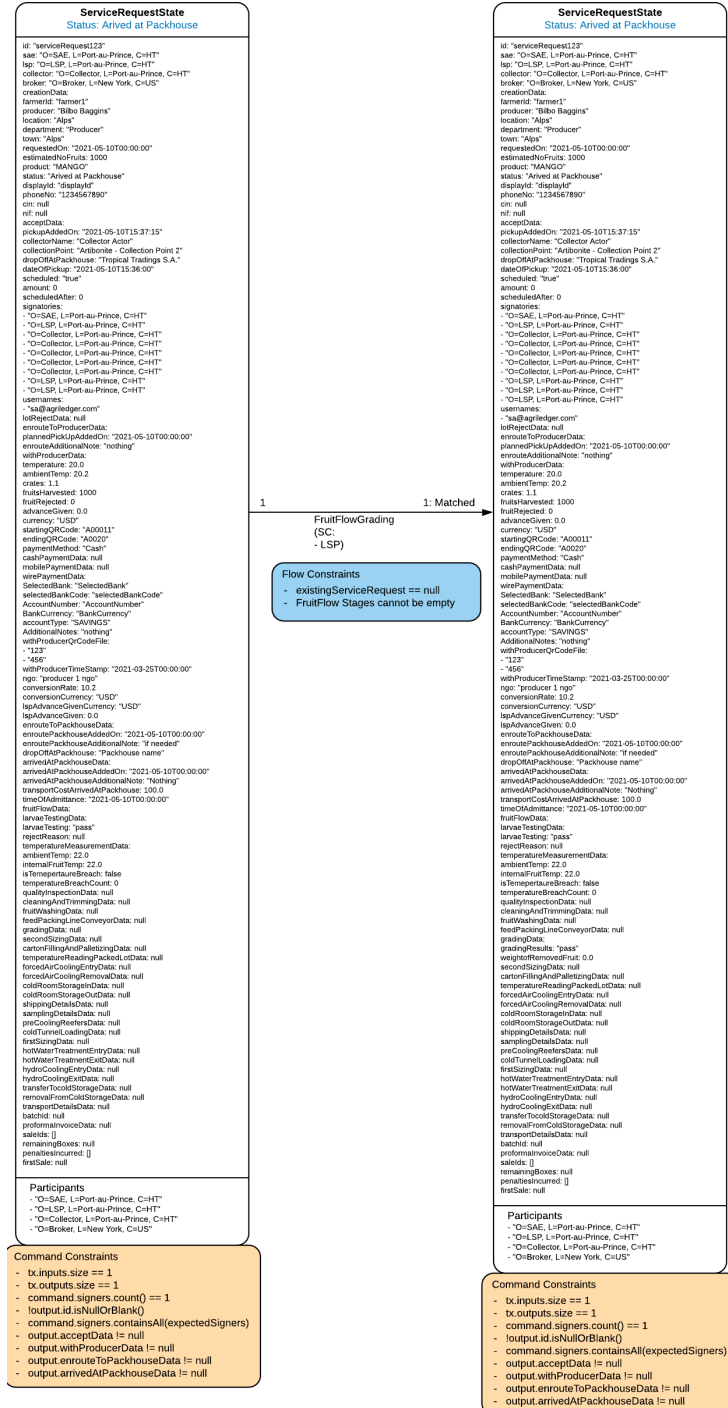
# 9. FruitFlow – Grading

Node: LSP

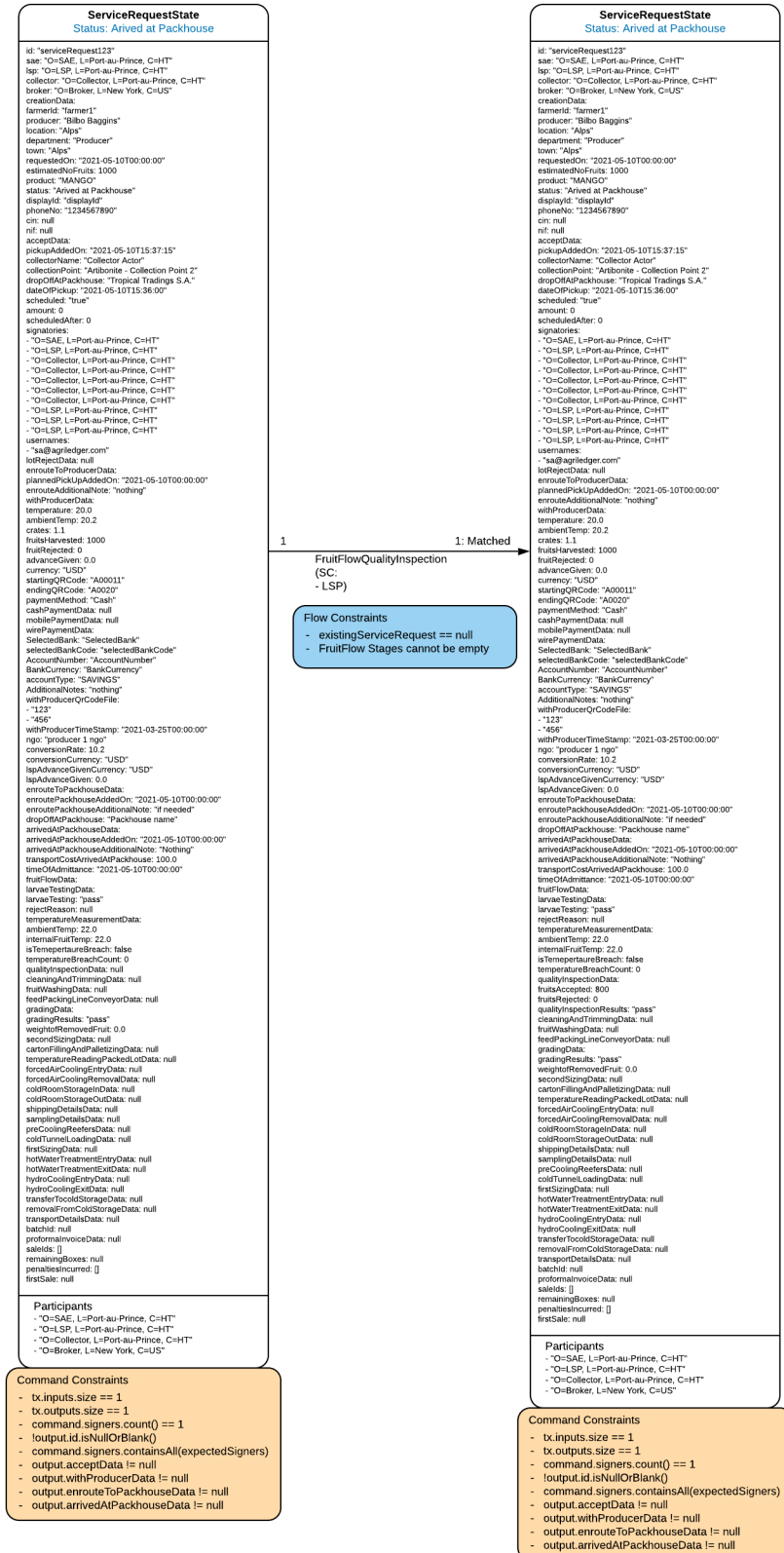
Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006



RPC Port: 10006



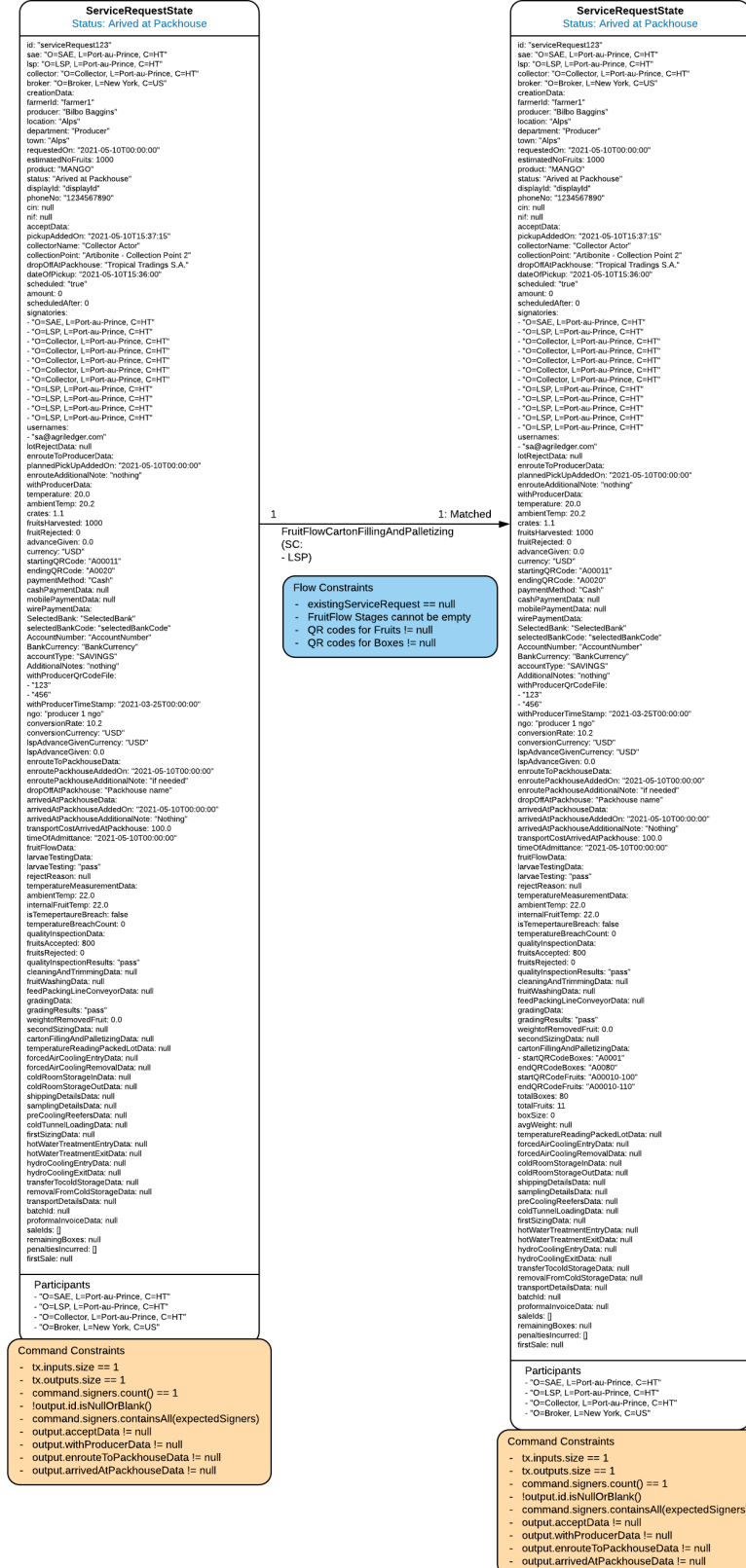
# 11. FruitFlow – Carton Filling and Palletizing

Node: LSP

Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006



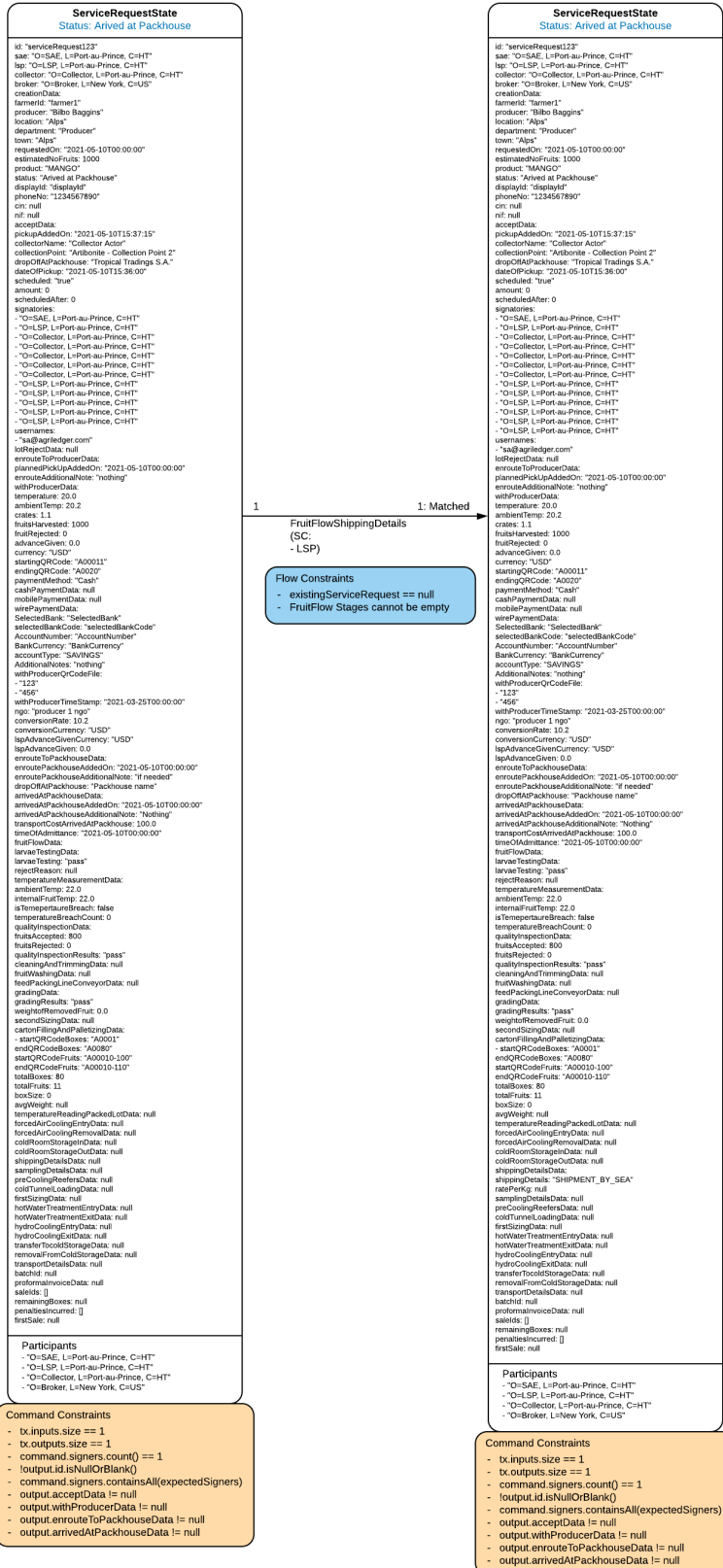
# 12. FruitFlow – Shipping Details

Node: LSP

Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006



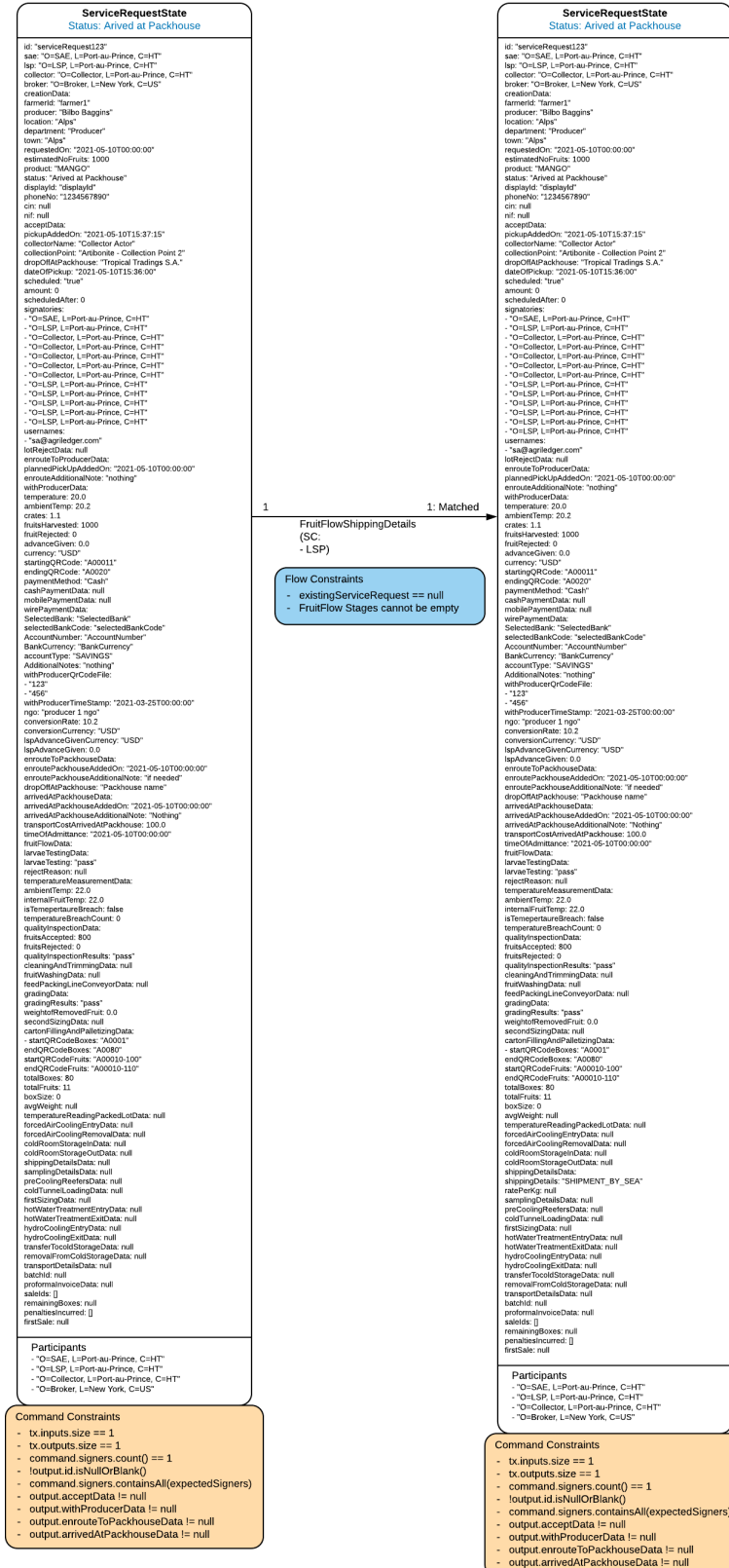
# 13. FruitFlow – Shipping Details

Node: LSP

Endpoint: POST - fruit-flow

Spring Host: 50007

RPC Port: 10006





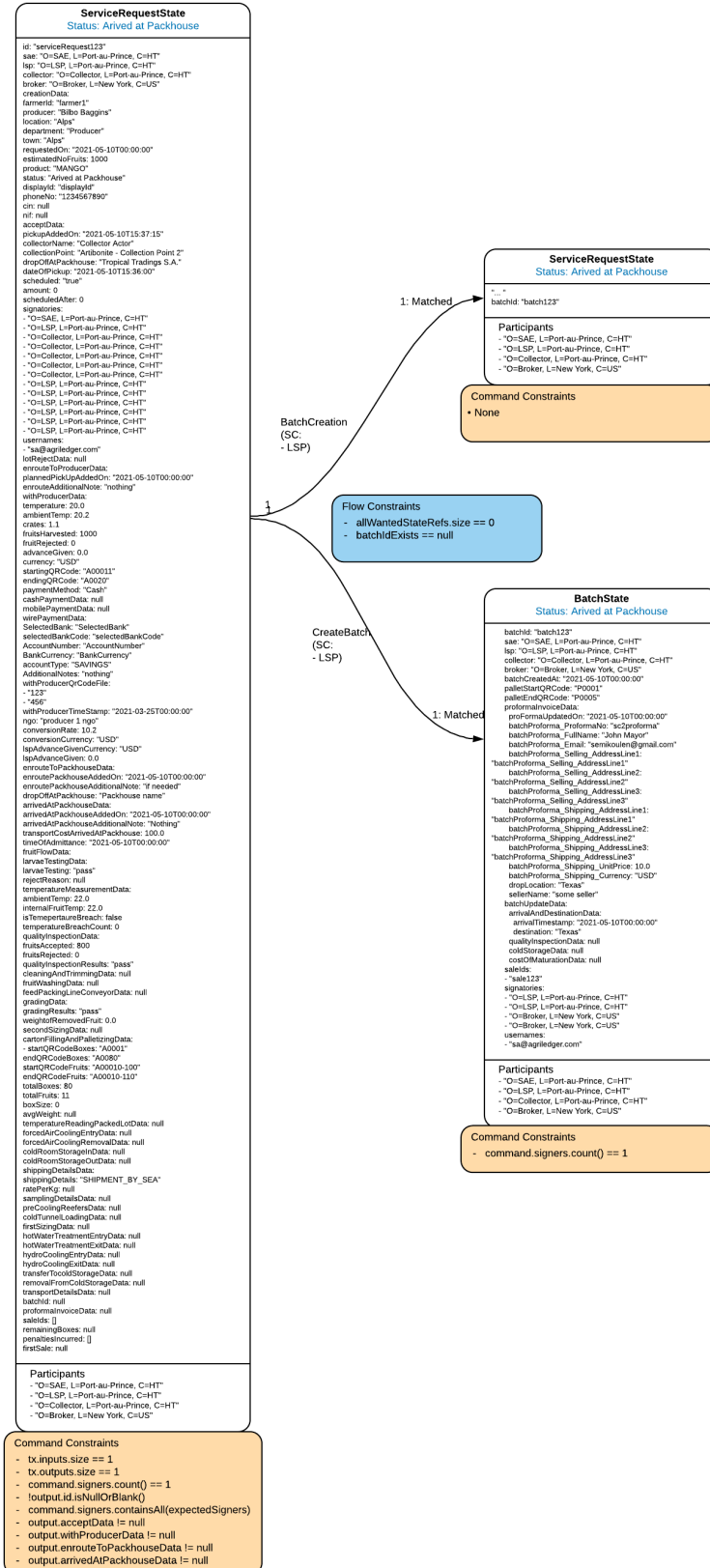
## 14. CreateNewBatchFlow

Node: LSP

Endpoint: POST - create-batch

Spring Host: 50007

RPC Port: 10006



## Other Discussion Topics

1. With the current solution for payments, will each payment have a state on the ledger? Would it be as a Tx and who creates the Tx? Can you elaborate & give details?

The *NewSalesFlow* is ran by a broker node to produce a *SalesState*. This state is defined to hold payment information properties such as *confirmPaymentData*, *paymentDistribution*, etc. As needed the Broker nodes can call flows that update this state such as *PaymentDistributions*, *Penalty Calculations*, *Sales Invoices*, etc. Each time this state is updated, it is done so through a Corda transaction.

2. Will payment confirmation be closing/consuming the corresponding open payment states? How would the system ensure that all confirmed payments are indeed what was originated from transactions recorded on the ledger?

The *ConfirmPaymentFlow* takes a *SalesState* as an input state and proposes an output which includes a *ConfirmationPaymentModel* object on the state. Here is a screenshot of some of the properties that are included in this update:

```
74 val outputState: SalesState = vaultData.copy(  
75     confirmPaymentData = reqParam?.let { it: ConfirmPaymentDTO  
76         ConfirmPaymentModel(  
77             confirmPaymentDate = LocalDateTime.parse(reqParam.confirmPaymentDate),  
78             netReceivables = reqParam.netReceivables.toFloat(),  
79             paymentNetReceivablesCurrency = reqParam.paymentNetReceivablesCurrency,  
80             wasFactored = reqParam.wasFactored.toBoolean(),  
81             amountFactored = reqParam.amountFactored?.toFloat(),  
82             factoringEntity = reqParam.factoringEntity,  
83             factoringAgent = reqParam.factoringAgent,  
84             factoringContactDetails = reqParam.factoringContactDetails,  
85             factoringCharges = reqParam.factoringCharges?.toFloat(),  
86             escrowAccountNo = reqParam.escrowAccountNo,  
87             brokerBankTransactionCurrency = reqParam.brokerBankTransactionCurrency,  
88             brokerBankTransactionFee = reqParam.brokerBankTransactionFee.toFloat(),  
89             netPayable = reqParam.netPayable.toFloat()
```

While the flow consumes an input state, the output state produced is never closed/exited and is capable of being updated further. If the state were to be closes/exited it would require an output state of null to prevent further changes.

3. How is the authorization managed to control which user role can call which API? For example, a harvest planner should not be allowed to call the "Update Sale" API.

Item number 9 in the 'Findings Summary' touches on a recommended approach to this as there currently is not any constraints in place to prevent certain flows to be executed by specific participants at this time.



4. What kind of security validations are/can be configured in the DevOps pipeline for nodes and APIs?

- [Corda RPC Security](#)
- [Corda Contract Constraints](#)
- [Monitoring](#)

5. Can Agriledger deliver the configuration of Azure and GCP resources as infrastructure-as-code, with build-in security wherever applicable?

[Question for Agriledger...](#)

6. In the document it is stated that blockchain nodes are run as windows services. But on the diagram "Exhibit 2" the VM that is displayed is a Linux machine. Running JVMs as windows services is not a known practice. Would it be better to run those as linux daemons on linux VMs? Linux VMs are cheaper to operate and faster to containerize.

[The most common Node deployment is on Azure VMs running Ubuntu.](#)

7. Have you looked into running each node as a container (Using Azure container services or app services) or a separate VM. Operation wise it would be easier and more redundant to containerize each service. VMs need constant upgrades, monitoring etc.

It is recommended to run one node per VM. Running multiple nodes on a VM could lead to a wider node outage if the VM were to crash where if it were just the one VM it would be limited.

[Docker containers are becoming more popular. Here is a blog post which discusses it at a high level.](#)

8. If public API service crashes whole solution will halt. It might be good to have two running instances of public api service behind a load balancer.

[Depending on your Cloud partner, high availability services such as load balancers and Hot/Cold deployments are usually offered for API endpoints.](#)

9. Is google firebase service really needed? There is no guarantee that it will be free in the future, operator has to deal with two cloud providers. Can't azure provide the same functionality. It's stated that number of transactions per second is not expected to be high. So is there a need for firebase?

[Question for Agriledger...](#)

# Conclusion

In walking through the code base and running the nodes, Spring Servers, and flows, the Agriledger CorDapp successfully demonstrated the lifecycle of its three states: *ServiceRequestState*, *BatchState*, and *SalesState*. While the flows successfully produced the expected state transitions it is important to note that they the payloads used in testing are what we would consider to be the “Happy Path” scenario.

As mentioned in item 3 'Conditional Flow Constraints', consider other situations that might not result in the status being set to the expected result such as "Arrived at Packhouse" when running the Temperature Measurement step in the FruitFlow (maybe the avocados were stored at 120 degrees and therefore the status should be set to "Expired Due to Temp" for example).

Besides giving the existing flows more dynamic functionality, the issue that was most concerning was the lack of logical checks in each contract's commands. Although applications will do most of these checks on the front end, the scenario where the front end is circumvented by a malicious actor must always be considered. In that case, the checks contained in the contract's commands are a ledger's last defence. Therefore, we strongly recommend adding as many checks as possible to ensure only perfect payloads can pass contract verification (only the properties you expect to change are changing and the ones that are changing are doing so in a way that you expect). Item number 4 'CreateBatchFlow' for example, shows a screenshot of a command that has no checks at all in the contract.

Overall Agriledger has had a great start in the development of the CorDapp and has produced a decent amount of functionality. The next steps are to provide the security to ensure the safety of the ledger's integrity and if possible, to generate a test suite that ensures that the application's functions are working/failing as expected. There are many situations and checks to consider in each of these flows, but I am confident that the Agriledger team can put the work in to get the application to where it needs to be for production.

The amount of information collated within this two week engagement is quite extensive, but of course it cannot be all encompassing. We recommend reviewing the material collated, along with the links to the public docs site that R3 is operating, where a lot more information exists.

If after reviewing this information there are still open questions, we recommend contacting R3 for inquiry into these additional topics. We are happy to help and want to make our customers as successful as they can be.



## About R3

R3 is an enterprise software firm working with a network of over 200 banks, financial institutions, regulators, trade associations, professional services firms and technology companies to develop Corda, its blockchain platform designed specifically for businesses. R3's global team of over 160 professionals in nine countries is supported by over 2,000 technology, financial, and legal experts drawn from its global member base.

Learn more at [r3.com](https://r3.com) and [corda.net](https://corda.net)

## Locations

### New York

11 West 42<sup>nd</sup> Street, 8<sup>th</sup> Floor, New York, NY 10036

### London

2 London Wall Palace, London, EC2Y 5AU

### Singapore

80 Robinson Road, #09-04 Singapore, 068898