Table of Contents

[Client 2](#_Toc166204859)

[Config 2](#_Toc166204860)

[GUI 2](#_Toc166204861)

[Service 3](#_Toc166204862)

[Commons 5](#_Toc166204863)

[DTOs 5](#_Toc166204864)

[Events 6](#_Toc166204865)

[Order Service 8](#_Toc166204866)

[Config 8](#_Toc166204867)

[Entity 8](#_Toc166204868)

[Service 8](#_Toc166204869)

[Inventory Service 10](#_Toc166204870)

[Config 10](#_Toc166204871)

[Entity 10](#_Toc166204872)

[Service 10](#_Toc166204873)

[Payment Service 11](#_Toc166204874)

[Config 11](#_Toc166204875)

[Entity 11](#_Toc166204876)

[Service 12](#_Toc166204877)

## Client

Client that interacts with the order microservice.

There can be multiple clients.

* Each client is associated with a sessionID
* In the GUI, the clients can specify any userID (can be the same userID for different clients)
* But it can be assumed that different client would specify a different userID (although the case where the same userID is used can be done)

### Config

#### ClientConsumerConfig

Configuration which specifies that the client consumes a CallbackEvent from the callback-event topic

#### ClientPublisherConfig

Specifies that the client would produce/publish a ClientEvent

### GUI

The GUI where the client can interact with the service (access point)

* Client have to enter all fields in numeric form:
  + UserID
  + EventID
  + Number of seats
* Ensuring that all fields are numeric is checked here
* However, how valid each of the fields are (e.g., whether the userID is a user that exists in the database) would be checked at each corresponding service

The GUI also has a text area (output box) that showcases either the outcome of the transaction, or a notification that one of the service might be down (if the reply back from the order service is not consumed within 10s)

* This timeout duration can be increased if needed

### Service

#### ClientConsumerService

This is where the client consumes CallbackEvent(s) from the callback-event topic

* It is noted that each client has a different group id to be able to consume all messages (rather than it being in the same group and being partitioned)
* Each client only consumes events meant for them through filtering out sessionIDs that does not match theirs

This consumer service also periodically checks (every 10s), whether a set time has passed since the last published event for this client

* If more than 10s passed since the last published event, and the client has not consumed any messages within that time, it signals that there might be a service failure somewhere
* This will then use the ClientService to notify the client

#### ClientPublisherService

Publishes the ClientEvent to the client-event topic

* ClientEvent includes the ClientRequestDTO, which has sessionID, UserID, EventID, and the number of seats wanted

#### ClientService

Includes the following methods:

* Launch GUI
* Update Client (output box)
* Stores the sessionID
* Stores the time which a message was last published by the client

ClientApplication

Checks whether the correct number of arguments was given (requires both ip address and port number), and verifies if the port given is a valid port number

* Yet to test: what happens if 2 clients specify the same port

If both of the above requirements are fulfilled, then it

* Sets the port for the client to be listening on
* Generates the sessionID
* Generates a unique groupID based on the sessionID
* Sets the groupID for this specific client

It then calls upon the ClientService to launch the GUI

## Commons

### DTOs

#### ClientRequestDTO

To be included in the ClientEvent

|  |  |
| --- | --- |
| sessionID |  |
| userID |  |
| eventID |  |
| seats |  |

#### OrderRequestDTO

To be included in the OrderEvent

|  |  |
| --- | --- |
| userID |  |
| eventID |  |
| seats |  |
| amount |  |
| orderID |  |

#### InventoryRequestDTO

To be included in the InventoryEvent

|  |  |
| --- | --- |
| userID |  |
| eventID |  |
| seats |  |
| amount |  |
| orderID |  |

#### PaymentRequestDTO

To be included in the PaymentEvent

|  |  |
| --- | --- |
| userID |  |
| amount |  |
| orderID |  |

#### OrderCallbackDTO

To be included in the CallbackEvent

|  |  |
| --- | --- |
| orderID |  |
| orderStatus |  |
| inventoryStatus |  |
| paymentStatus |  |

### Events

#### Event (Interface)

#### ClientEvent

Includes: eventID, date, clientRequestDTO

Sent from: client

To topic: client-event

Consumed by: order service

#### OrderEvent

Includes: eventID, date, orderRequestDTO, orderStatus (enum)

Sent from: order service

To topic: order-event

Consumed by: inventory service

#### InventoryEvent

Includes: eventID, date, inventoryRequestDTO, inventoryStatus (enum)

Sent from: inventory service

To topic: inventory-event OR event-updates

* sent to event-updates if inventor check fails

Consumed by: payment service OR order service

#### PaymentEvent

Includes: eventID, date, paymentRequestDTO, paymentStatus (enum)

Sent from: payment service

To topic: event-updates

Consumed by: order service

#### CallbackEvent

Includes: eventID, date, sessionID, orderCallbackDTO

Sent from: order service

To topic: callback-event

Consumed by: client

## Order Service

### Config

#### OrderConsumerConfig

Specifies that the order microservice consumes a ClientEvent from the client-event topic

Specifies that the order microservice consumes an Event (general) from the order-updates topic

* Consumes a general event to accommodate for the fact that either InventoryEvent or PaymentEvent can be sent to the order-updates topic

#### OrderPublisherConfig

Specifies that the client can produce/publish a ClientEvent and a CallbackEvent

### Entity

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| orderID | userID | sessionID | eventID | price | seats | order status | inventory status | payment status |
|  |  |  |  |  |  |  |  |  |

To be stored in the repository (OrderRepo interface)

### Service

#### OrderConsumerService

From client-event, consumes ClientEvent, and starts the order

* Starting the order includes creating the orderRequestDTO, and calling the order service to actually create an order

From order-updates, consumes Event

* Processes this event, which handles the InventoryEvent and PaymentEvent separately – calls different methods based on the instance of the Event received/consumed
* Based on the type of event, different methods in OrderService is called to update the order

*Note: currently, the ticket price for all events is $10 LOL*

#### OrderPublisherService

Publishes the events to the different topics accordingly (order-event or callback-event)

#### OrderService

Creates the order, which saves the OrderEntity to the repo, and then publishes calls the method to publish the OrderEvent to the order-event topic with the orderStatus: ORDER\_CREATED

Updates the order based on consumption of InventoryEvent

* If it received an InventoryEvent, it can only mean that the inventory check failed, or there’s no such event:
  + orderStatus: ORDER\_FAILED
  + inventoryStatus: INVENTORY\_CHECK\_FAILED or NO\_EVENT
  + paymentStatus: null
* Then publishes this as a callbackEvent to callback-event topic

Updates the order based on consumption of PaymentEvent

* If it received a PaymentEvent, it could have either succeeded or failed, and it can only mean that the inventory check is successful
* If payment successful:
  + orderStatus: ORDER\_SUCCESS
  + inventoryStatus: INVENTORY\_CHECK\_SUCCESS
  + paymentStatus: PAYMENT\_SUCCESS
* If payment unsuccessful aka not enough user balance (or no such user)
  + orderStatus: ORDER\_FAILED
  + inventoryStatus: INVENTORY\_CHECK\_SUCCESS
  + paymentStatus: PAYMENT\_FAILED or USER\_NOT\_FOUND

In both update methods, the order entity with its updated status is saved back into the repo, and is then published to the topics:

* Success: to callback-event
* Failed: to order-event AND callback-event
  + For inventory service to consume and perform compensation

**!!! But with the payment unsuccessful method, an additional publishing back to the order-event topic is needed for the inventory service to consume and compensate**

* **TO BE TESTED!!!!**
* **BUT what if the inventory service fails – it will consume this later on when it gets back – is Apache Kafka a reliable channel?**

## Inventory Service

### Config

#### InventoryConsumerConfig

Specifies that the inventory microservice consumes a OrderEvent from the order-event topic

#### InventoryPublisherConfig

Specifies that the inventory microservice will publish an InventoryEvent

### Entity

#### EventInventory

|  |  |
| --- | --- |
| eventID | seatsAvail |
|  |  |

#### InventoryTransaction

|  |  |  |
| --- | --- | --- |
| orderID | eventID | seats |
|  |  |  |

Adds a transaction when checking inventory, deleted if insufficient inventory available

### Service

#### InventoryConsumerService

Consumes order event, and upon consuming, validates inventory

* If order status is created, it will create a new order event in the validation service, and then publish the inventory event returned with InventoryEventPublisher
* If any other status (i.e., failed), it will cancel the order event, and peform the compensation action

#### InventoryEventPublisher

If the inventory check is successful, it will publish to the inventory-event topic for the payment service to consume

Otherwise, it will publish to the order-updates topic, indicating that it has failed

#### InventoryValidationService

If the order status is created, it creates a new order event.

* If the event exists:
  + Deduct from inventory if there is sufficient inventory available, and the inventory status will reflect success
  + Otherwise, no deduction is done, and the inventory status will reflect a failure
* If the event does not exist,
  + No deduction is done, and the inventory status will reflect that there is no such event

If the event needs to be cancelled (bc payment failed for example):

* Finds the inventory transaction from the repo, and if it exists, it will add the number of seats back from that order into the eventInventory repo
* Then, delete this record from the repository (ensure that this operation is idempotent)

## Payment Service

### Config

#### PaymentConsumerConfig

Specifies that the payment microservice consumes a InventoryEvent from the inventory-event topic

#### PaymentPublisherConfig

Specifies that it will produce a PaymentEvent

### Entity

#### UserBalance

|  |  |
| --- | --- |
| userID | balance |
|  |  |

#### UserTransaction

|  |  |  |
| --- | --- | --- |
| orderID | userID | amount |
|  |  |  |

### Service

#### PaymentConsumerService

Consumes InventoryEvent from inventory-event, and validates the balance

* If inventory check is successful, it publishes the paymentEvent
* If inventory check is not successful, then it cancels this event (although it won’t reach this case, unless the user actually cancel their booking, which is not implemented here)
  + Because if inventory check is not successful, it won’t be forwarded here in the first place

#### PaymentEventPublisher

Publishes to the order-updates topic only

#### PaymentValidationService

When creating a new inventory event, it checks whether the user has sufficient balance to actually book/reserve the tickets

* Before that though, it checks whether such a user exists in the userBalance repo
* If user exists:
  + If sufficient balance, sets the new balance of the user by deducting amount of transaction from the balance, then return the payment event with the payment status reflecting a success, to the consumer service which will then publish this event
  + Otherwise, it will return a payment event reflecting failure in payment
* If user does not exist:
  + It will return a payment event that indicates that there is no such user