Step 4

Modes of Composition in functional Scala programming

Ben Hutchison
Lambdajam Online 2020

Step 4: Processing an Available Order

- In step 3, we built a check that filtered out orders that included Skus marked as not available to the Customer's Region.

- In step 4, we'll implement the logic for interacting with the Inventory, which tracks the items available in the warehouses or supply

    - the default "happy path" is where the Skus in the Order are deducted from the Inventory, then message is sent to Dispatch the order (eg to workers who pick the items to fulfil)

    - What if, for whatever reason, there are insufficient items in inventory to fulfil the order? In this case, a Backorder is generated to trigger refill of the inventory from suppliers for those items that are short. All goods are to be returned to inventory; ie orders are not part-fulfilled.

Inventory

- The Inventory support two operations.
    - Taking from the Inventory either succeeds, or if there is insufficient stock it fails with an InsufficientStock record that details the shortfall
    - In this exercise, the put operation is only used when a order's items need to be returned, because some part of the order was unavailable
        - "Application-level transaction"

```
def inventoryTake(skuQty: SkuQuantity): F[Either[InsufficientStock, SkuQuantity]]

def inventoryPut(skuQty: SkuQuantity): F[Unit]
```

Dispatched and Backorder

- Order Processing results in one of two messages

- Dispatched events are the “happy path”: all items are available, the stock is deducted and it is sent for packing
    - Dispatched(order: CustomerOrder, timestamp: Instant, fulfillmentId: UUID)

- Backorder events describe the Skus that are short stock and must be backordered to fulfill the order
    - Backorder(required: NonEmptyChain[SkuQuantity], order: CustomerOrder, timestamp: Instant)

UuidRef

- UUIDs are random 128bit IDs.
  - Good way to generate unique, non-sequential, non-guessable IDs
- To generate them requires a random seed
- The seed is stored in a cats.effect.concurrent.Ref
  - A mutable store that can be safely accessed concurrently

Zooming in on the problem

- This problem has been split out into 5 parts
  - Use type-driven development techniques to builds pieces of the solution, then combine
  - Start bottom-up with these 3 building blocks:

def **insufficientsAndTaken**(takes: NonEmptyChain[Either[InsufficientStock, SkuQuantity]]): Option[(NonEmptyChain[InsufficientStock], Chain[SkuQuantity])]

def **dispatch**[F[_]: Sync: Clock: UuidRef](order: CustomerOrder): F[Dispatched]

def **backorder**[F[_]: Sync: Clock](insufficientStocks: NonEmptyChain[InsufficientStock], order: CustomerOrder): F[Backorder]

Zooming in on the problem

- This problem has been split out into 5 parts
  - Use type-driven development techniques to builds pieces of the solution, then combine
  - Start bottom-up with these 3 building blocks:

def **insufficientsAndTaken**(takes: NonEmptyChain[Either[InsufficientStock, SkuQuantity]]): Option[(NonEmptyChain[InsufficientStock], Chain[SkuQuantity])]

def **dispatch**[F[_]: Sync: Clock: UuidRef](c rOrder): F[Dispatched]

def **backorder**[F[_]: Sync: C er): F[Backorder]

Cats operation **separate** is the key to splitting the takes into Left and Right groups
Convert it **toChain** beforehand

Zooming in on the problem

- This problem ha
  - Use type-drive
  - Start bottom-u

JavaTime[F].getInstant gets a timestamp
UuidSeed.nextUuid[F] gets a UUID
Put effectful expressions in a tuple and use **mapN** to combine the results

def **insufficientsAndTaken**(takes: NonE ither[InsufficientStock, SkuQuantity]]): Option[(NonEmptyChain[InsufficientSto ain[SkuQuantity])]

def **dispatch**[F[_]: Sync: Clock: UuidRef](order: CustomerOrder): F[Dispatched]

def **backorder**[F[_]: Sync: Clock](insufficientStocks: NonEmptyChain[InsufficientStock], order: CustomerOrder): F[Backorder]

Zooming in on the problem

- dispatchElseBackorder should then use **insufficientsAndTaken, dispatch** and **backorder** in its implementation

```
def dispatchElseBackorder[F[_]: Sync: Parallel: Clock: UuidRef: Inventory](order: CustomerOrder):
  F[Either[(Backorder, Chain[SkuQuantity]), Dispatched]]
```

Zooming in on the problem

- **dispatchElseBackorder** should then use **insufficientsAndTaken, dispatch** and **backorder** in its implementation

def **dispatchElseBackorder**[F[_]: Sync: Parallel: Clock: UuidRef: Inventory](order: CustomerOrder):

  F[Either[(Backorder, Chain[SkuQuantity]), Dispatched]]

parTraverse the items in the order to **inventoryTake** on each of them.
Then use **insufficientsAndTaken** to give a result in a convenient form to pattern match on, with cases for **dispatch** and **backorder**

Zooming in on the problem

- **processAvailableOrder** should use **dispatchElseBackorder** then pattern match on the result to publish appropriate messages

```
def processAvailableOrder[F[_]: Sync: Parallel: Clock: UuidRef: Inventory: Publish]
  (order: CustomerOrder): F[Unit]
```