Step 3

Modes of Composition in functional Scala programming

Ben Hutchison
Lambdajam Online 2020

Step 3: Processing an Unavailable Order

- Some Skus cannot be delivered to particular regions due to regulation or delivery constraints

- If any Skus in the Order are marked as unavailable in the Customer's Region
  - the order should not proceed to fulfilment
  - An **Unavailable** event should be sent to the Unavailable topic

- Otherwise, proceed with order processing, which is stubbed out for this step but will be tackled in step 4

Zooming in on the problem

```
def processCustomerOrder[F[_]: Functor: Sync: Parallel: Clock: UuidRef: Inventory: Publish]( order: CustomerOrder): F[Unit] = {

 val nonAvailableSkus: Chain[Sku] = ???

 if (nonAvailableSkus.isEmpty)

   ???

 else {

   ???

 }

}
```

Zooming in on the problem

```
def processCustomerOrder[F[_]: Functor: Sync: Parallel: Clock: UuidRef: Inventory: Publish]( order: CustomerOrder): F[Unit] = {
 val nonAvailableSkus: Chain[Sku] = ???
 if (nonAvailableSkus.isEmpty)
   processAvailableOrder[F](order)
 else {
   ???
 }
}
```

Clock and UuidRef effects are new in this step

Clock lets us sample current time

UuidRef is some state used for generating random UUIDs. It is only needed to pass through to processAvailableOrder

Zooming in on the problem

```
def processCustomerOrder[F[_]: Functor: Sync: Parallel: Clock: UuidRef: Inventory: Publish]( order: CustomerOrder): F[Unit] = {

 val nonAvailableSkus: Chain[Sku] = ???

 if (nonAvailableSkus.isEmpty)

   processAvailableOrder[F](order)

 else {

   ???

 }

}
```

making a list of non available Skus is just a matter of walking through the Skus in the order matching against the customer's Region

Its a pure, non-effectful computation, in that the result isn't wrapped up in an F-effect

Zooming in on the problem

```
def processCustomerOrder[F[_]: Functor: Sync: Parallel: Clock: UuidRef: Inventory: Publish]( order: CustomerOrder): F[Unit] = {
 val nonAvailableSkus: Chain[Sku] = ???
NonEmptySet.fromSet(SortedSet.from(nonAvailableSkus.iterator)) match {
 case None =>
   processAvailableOrder[F](order)
 case Some(nonAvailableSet) =>
   ???
}
}
```

If there are no Skus that are nonAvailable, proceed to process the order

Zooming in on the problem

```
def processCustomerOrder[F[_]: Functor: Sync: Parallel: Clock: UuidRef: Inventory: Publish]( order: CustomerOrder): F[Unit] = {
 val nonAvailableSkus: Chain[Sku] = ???
NonEmptySet.fromSet(SortedSet.from(nonAvailableSkus.iterator)) match {
 case None =>
   processAvailableOrder[F](order)
 case Some(nonAvailableSet) =>
   ???
}
}
```

Otherwise, we need to:
Get the current time, build an Unavailable object, and publish the object's bytes to the Unavailable topic

Getting the Time

- Cats Effect defines a Clock effect for accessing the time
  - low level: milliseconds since 1970 as a Long
- We use a small library cats-effect-time by Chris Davenport to convert the clock into Java Time types
  - JavaTime[F].getInstant returns us a F[Instant]

Getting a NonEmptySet of non-available Skus

- NonEmptySet is a Cats type
  - Wants proof on construction that it has at least one element
- When we construct from a regular, potentially empty set, we get back an Option[NonEmptySet]
  - **NonEmptySet.fromSet(SortedSet.from(nonAvailableSkus.iterator))**

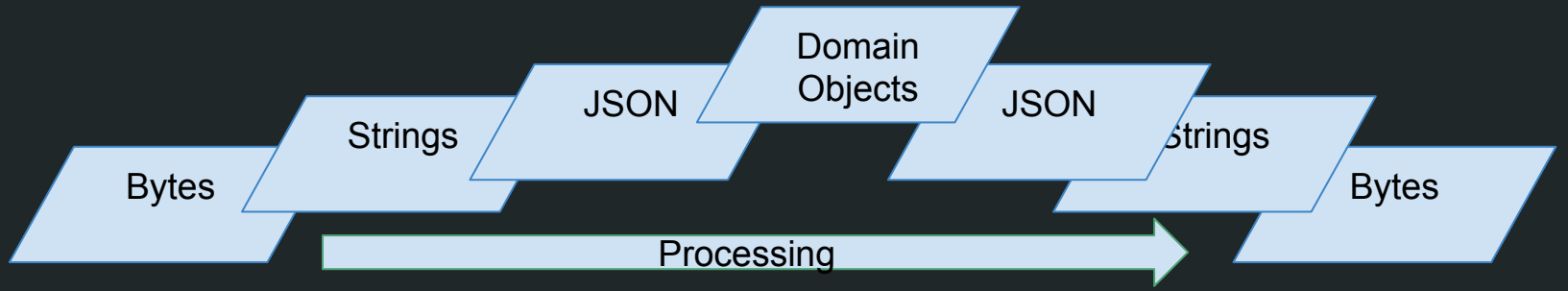
Publishing an Unavailable message

- The Publish capability supports publishing messages, as **Array[Byte]** to named topics
  - Topic names pre-defined on the **Topic** object
- Converting a message *to* bytes is easier than *from* bytes
  - Travelling “downhill” from a structured domain object representation to less structured String and Array[Byte] representations
  - **asJson** converts to JSON AST using an Encoder
  - **toString** converts to String
  - **getBytes** converts to Bytes

- A JSON **Codec**, a **Decoder** + **Encoder**, is an example of a function paired with it’s inverse
  - Travel back and forth between JSON and object representation

Modes of Composition

- An effectful value is internally composed of layers
  - Different rules or invariants apply in each layer
- The typical structure sees weakly typed layers on the outer rings of an application and one to many more strongly typed interior layers
  - Data representation is different in each layer



Modes of Composition

- *An effectful value and it's (approximate) inverse are paired together to create an effectful isomorphism*
  - *Resources, Codecs, Functional Optics, Sagas*

- The inverse of a function reverses, or undoes, its effect
- When paired, a function and its inverse enable a program to "travel" between a mode where the function applies, and one where its doesn't
  - Turns out useful in a broad variety of superficially unrelated situations