AppsFlyer Engineering

Help the compiler help you

Leveraging Scala's implicits for code safety

Scala meetup - September 2022

What will we talk about?

01

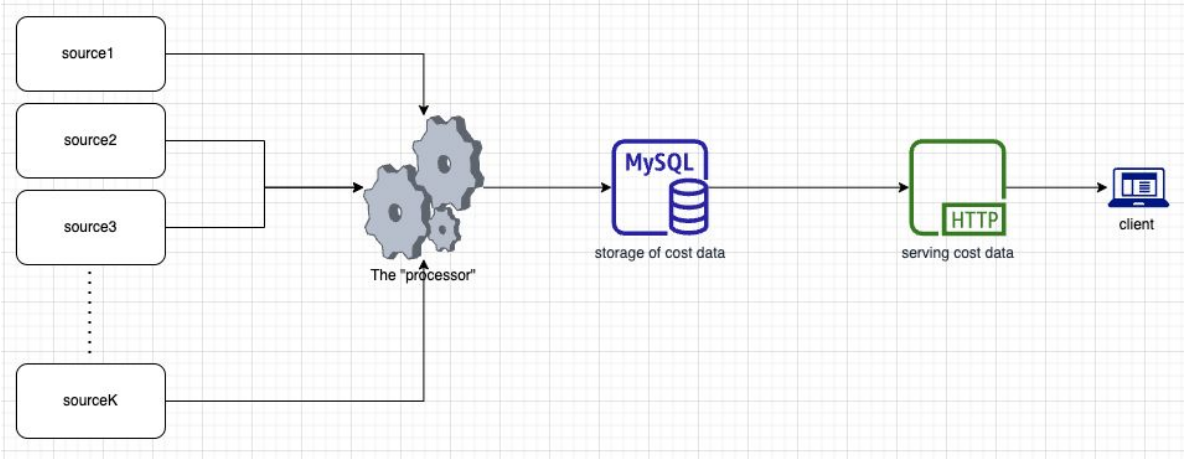
"TODO or not TODO"?
A day in the life of a SW engineer

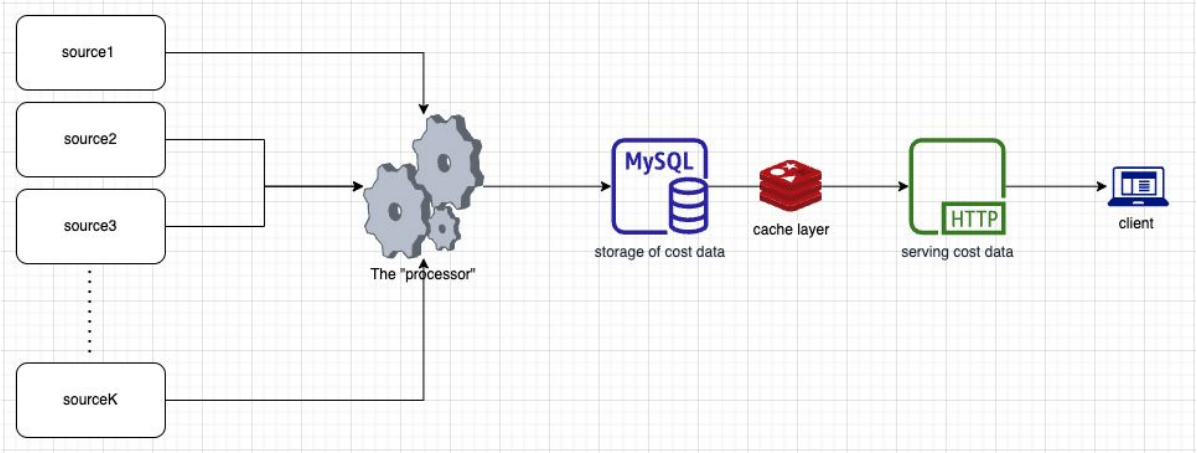
02

"let's make it right, shall we?"

- Chapter 1 -

“TODO or not TODO”?
A day in the life of a SW engineer


source1
source2
source3
sourceK
The "processor"
MySQL
storage of cost data
HTTP
serving cost data
client


source1
source2
source3
sourceK
The "processor"
MySQL
storage of cost data
cache layer
HTTP
serving cost data
client

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
sealed trait CostModel {
  val cost: Double
  val date: LocalDate
  val accountId: AccountId

  override def toString: String =
    """
      |cost: $cost
      |date: $date
      |accountId: $accountId
      |""".stripMargin
}

//cost per install
case class CPI(cost: Double,
               date: LocalDate,
               accountId: AccountId,
               countryCode: String)
    extends CostModel

//cost per action
case class CPA(cost: Double,
               date: LocalDate,
               accountId: AccountId,
               channel: String)
    extends CostModel
```

```
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

TODO:

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

TODO:

- ➢ create redis client
- ➢ Generate for each model, a proper redis key
- ➢ On given request, check in redis by key. If not found, ask DB
- ➢ Later on, store the result in redis, with the generated key

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {

  println("cost serving http server has started..")

  startHttpServer()
}
```

```scala
def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
  for {
    costResult ← getCostFromDB(request.accountId, request.date)
    _ ← logger.info(s"received: $costResult from db")
  } yield buildRespons(costResult)
}

private def getCostFromDB(accountId: AccountId,
                          date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}
```

```scala
def main(args: Array[String]): Unit = {

  println("cost serving http server has started..")

  startHttpServer()
}
```

```scala
def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
  for {
    costResult ← getCostFromDB(request.accountId, request.date)
    _ ← logger.info(s"received: $costResult from db")
  } yield buildRespons(costResult)
}

private def getCostFromDB(accountId: AccountId,
                          date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}
```

```scala
def main(args: Array[String]): Unit = {

  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
  for {
    costResult ← getCostFromDB(request.accountId, request.date)
    _ ← logger.info(s"received: $costResult from db")
  } yield buildRespons(costResult)
}

private def getCostFromDB(accountId: AccountId,
                          date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)

   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)

   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)

   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}
```

```scala
private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
```

```scala
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }
```

```scala
private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {

    println("cost serving http server has started..")

    startHttpServer()
  }

  def handleCostRequest(request: CostHttpRequst): IO[HttpResponse] = {
    for {
      costResult ← getCostFromDB(request.accountId, request.date)
      _ ← logger.info(s"received: $costResult from db")
    } yield buildRespons(costResult)
  }

  private def getCostFromDB(accountId: AccountId,
                            date: LocalDate): IO[CostModel] = {
```

```scala
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)

   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

We run our code and...

```
Exception in thread "main" java.lang.Exception Create breakpoint : NOAUTH Authentication required.
    at com.redis.Reply$$anonfun$errReply$1.applyOrElse(RedisProtocol.scala:132)
    at com.redis.Reply$$anonfun$errReply$1.applyOrElse(RedisProtocol.scala:131)
    at scala.runtime.AbstractPartialFunction.apply(AbstractPartialFunction.scala:38)
    at com.redis.Reply$$anonfun$bulkReply$1.applyOrElse(RedisProtocol.scala:91)
    at com.redis.Reply$$anonfun$bulkReply$1.applyOrElse(RedisProtocol.scala:91)
    at scala.PartialFunction$OrElse.apply(PartialFunction.scala:172)
    at com.redis.Reply.receive(RedisProtocol.scala:152)
    at com.redis.Reply.receive$(RedisProtocol.scala:148)
    at com.redis.Redis.receive(RedisClient.scala:34)
    at com.redis.R.asBulk(RedisProtocol.scala:255)
    at com.redis.R.asBulk$(RedisProtocol.scala:255)
    at com.redis.Redis.asBulk(RedisClient.scala:34)
    at com.redis.StringOperations.$anonfun$get$1(StringOperations.scala:24)
    at com.redis.Redis.send(RedisClient.scala:45)
    at com.redis.StringOperations.get(StringOperations.scala:24)
    at com.redis.StringOperations.get$(StringOperations.scala:23)
    at com.redis.RedisCommand.get(RedisClient.scala:96)
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}


private def authenticateRedisClient(redisClient: RedisClient,
                                    secret: String): Unit = {
  val res = redisClient.auth(secret)

  if (!res){
    throw new Exception("can't authenticate redis!")
  }
}
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)

  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)
  authenticateRedis(redisClient, "secret")
  println("cost serving http server has started..")

  startHttpServer()
}


private def authenticateRedisClient(redisClient: RedisClient,
                                    secret: String): Unit = {
  val res = redisClient.auth(secret)

  if (!res){
    throw new Exception("can't authenticate redis!")
  }
}
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What are the possible problems with this implementation?

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What are the possible problems with this implementation?

1. Should we wait for redis usage to authenticate and throw? Or put it all in main?

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What are the possible problems with this implementation?

1. Should we wait for redis usage to authenticate and throw? Or put it all in `main`?

2. `getCostFromCache`, `setToCache` methods receives a `redisClient`. No indication of it's "authentication" state!

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What are the possible problems with this implementation?

1. Should we wait for redis usage to authenticate and throw? Or put it all in `main`?

2. `getCostFromCache`, `setToCache` methods receives a `redisClient`. No indication of it's "authentication" state!

3. Generation of the "redis key"

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What are the possible problems with this implementation?

1. Should we wait for redis usage to authenticate and throw? Or put it all in `main`?
2. `getCostFromCache`, `setToCache` methods receives a `redisClient`. No indication of it's "authentication" state!
3. Generation of the "redis key"
4. How to test methods which require `RedisClient`?

- Chapter 2 -

“let's make it right, shall we?”

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
  implicit val redisClient: RedisClient = new RedisClient("host", 6379)
  authenticateRedis(redisClient, "secret")
  println("cost serving http server has started..")

  startHttpServer()
}

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
implicit val redisClient: RedisClient = new RedisClient("host", 6379)
```

```scala
implicit val redisClient: RedisClient = new RedisClient("host", 6379)
```

TODO:

```scala
implicit val redisClient: RedisClient = new RedisClient("host", 6379)
```

TODO:

- Allow testability to methods using `RedisClient`
- “Embed” somehow the “authentication” state in the `RedisClient`
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

TODO:

- Allow testability to methods using `RedisClient`

TODO:

- Allow testability to methods using `RedisClient`

Solution:

- “Tagless final” approach

```scala
trait RedisOpsT {
  def authenticate(secretKey: String): Unit
  def get(key: String): Option[String]
  def set(key: String, value: Any): Boolean
}
```

TODO:

- Allow testability to methods using `RedisClient`

Solution:

- “Tagless final” approach
  a. define an “algebra”

```scala
trait RedisOpsT {

  def authenticate(secretKey: String): Unit

  def get(key: String): Option[String]

  def set(key: String, value: Any): Boolean

}

case class RedisOpsSingleThread(redisClient: RedisClient) extends RedisOpsT {
  override def authenticate(secretKey: String): Unit = {
    val res = redisClient.auth(secretKey)

    //TODO return as value. Try / Either or others..
    if (!res) {
      throw new Exception(f"Could not authenticate redis")
    }
  }

  override def get(key: String): Option[String] = {
    redisClient.get[String](key)
  }

  override def set(key: String, value: Any): Boolean =
    redisClient.set(key, value)
}
```

TODO:

- Allow testability to methods using `RedisClient`

Solution:

- “Tagless final” approach
  a. define an “algebra”
  b. implement “interpreters”

```scala
trait RedisOpsT {

  def authenticate(secretKey: String): Unit

  def get(key: String): Option[String]

  def set(key: String, value: Any): Boolean

}

case class RedisOpsSingleThread(redisClient: RedisClient) extends RedisOpsT {
  override def authenticate(secretKey: String): Unit = {
    val res = redisClient.auth(secretKey)

    //TODO return as value. Try / Either or others..
    if (!res) {
      throw new Exception(f"Could not authenticate redis")
    }
  }

  override def get(key: String): Option[String] = {
    redisClient.get[String](key)
  }

  override def set(key: String, value: Any): Boolean =
    redisClient.set(key, value)
}
object RedisOpsSingleThread {
  def apply(host: String, port: Int): RedisOpsSingleThread = {
    val redisClient: RedisClient =
      new RedisClient(host, port)
    RedisOpsSingleThread(redisClient)
  }
}
```

TODO:

- Allow testability to methods using `RedisClient`

Solution:

- “Tagless final” approach
  a. define an “algebra”
  b. implement “interpreters”

```scala
trait RedisOpsT {

  def authenticate(secretKey: String): Unit

  def get(key: String): Option[String]

  def set(key: String, value: Any): Boolean

}

case class RedisOpsSingleThread(redisClient: RedisClient) extends RedisOpsT {
  override def authenticate(secretKey: String): Unit = {
    val res = redisClient.auth(secretKey)

    //TODO return as value. Try / Either or others..
    if (!res) {
      throw new Exception(f"Could not authenticate redis")
    }
  }

  override def get(key: String): Option[String] = {
    redisClient.get[String](key)
  }

  override def set(key: String, value: Any): Boolean =
    redisClient.set(key, value)
}
object RedisOpsSingleThread {
  def apply(host: String, port: Int): RedisOpsSingleThread = {
    val redisClient: RedisClient =
      new RedisClient(host, port)
    RedisOpsSingleThread(redisClient)
  }
}
```

TODO:

- Allow testability to methods using RedisClient

```scala
val clientForTesting = new RedisOpsT {
  override def authenticate(secret: String): Unit =
    println("you are set to go")

  override def get(key: String): Option[String] =
    Some("42") // why 42?

  override def set(key: String, value: Any): Boolean = {
    println(s"setting key: $key with value $value")
    true
  }
}
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- Allow testability to methods using RedisClient
- “Embed” somehow the “authentication” state in the RedisClient
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- ✓ Allow testability to methods using RedisClient
- “Embed” somehow the “authentication” state in the RedisClient
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- ✓ Allow testability to methods using RedisClient
- “Embed” somehow the “authentication” state in the RedisClient
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

TODO:

- “Embed” somehow the “authentication” state in the `RedisClient`

TODO:

- “Embed” somehow the “authentication” state in the `RedisClient`

Solution:

- Phantom Types!

```scala
trait RedisOpsT {
  def authenticate(secretKey: String): Unit
  def get(key: String): Option[String]
  def set(key: String, value: Any): Boolean
}
```

TODO:

- “Embed” somehow the “authentication” state in the `RedisClient`

Solution:

- Phantom Types!

```scala
trait RedisOpsT {

  def authenticate(secretKey: String): Unit

  def get(key: String): Option[String]

  def set(key: String, value: Any): Boolean

}

object RedisOpsT {
  //We use here "State" as a "Phantom type" to ensure
  //'get' or 'set' arn't called before authenticating
  sealed trait State

  object State {
    sealed trait Authenticated extends State

    sealed trait UnAuthenticated extends State
  }
}
```

TODO:

- “Embed” somehow the “authentication” state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the “states” as `sealed trait` inside the `RedisOpsT` companion object

```scala
trait RedisOpsT {

  def authenticate(secretKey: String): Unit

  def get(key: String): Option[String]

  def set(key: String, value: Any): Boolean

}

object RedisOpsT {
  //We use here "State" as a "Phantom type" to ensure
  //'get' or 'set' arn't called before authenticating
  sealed trait State

  object State {
    sealed trait Authenticated extends State

    sealed trait UnAuthenticated extends State
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the "states" as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait

```scala
trait RedisOpsT[S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[S <: State.Authenticated](key: String): Option[String]

  def set[S <: State.Authenticated](key: String, value: Any): Boolean

}

object RedisOpsT {
  //We use here "State" as a "Phantom type" to ensure
  //'get' or 'set' arn't called before authenticating
  sealed trait State

  object State {
    sealed trait Authenticated extends State

    sealed trait UnAuthenticated extends State
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the "states" as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait

```scala
trait RedisOpsT[S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[S <: State.Authenticated](key: String): Option[String]

  def set[S <: State.Authenticated](key: String, value: Any): Boolean

}

object RedisOpsT {
  //We use here "State" as a "Phantom type" to ensure
  //'get' or 'set' arn't called before authenticating
  sealed trait State

  object State {
    sealed trait Authenticated extends State

    sealed trait UnAuthenticated extends State
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the "states" as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait

```scala
trait RedisOpsT[S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[S <: State.Authenticated](key: String): Option[String]

  def set[S <: State.Authenticated](key: String, value: Any): Boolean

}

object RedisOpsT {
  //We use here "State" as a "Phantom type" to ensure
  //'get' or 'set' arn't called before authenticating
  sealed trait State

  object State {
    sealed trait Authenticated extends State

    sealed trait UnAuthenticated extends State
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the "states" as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait
  - Align in all implementations

TODO:

- “Embed” somehow the “authentication” state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the “states” as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait
  - Align in all implementations

```scala
case class RedisOpsSingleThread(redisClient: RedisClient) extends RedisOpsT {
  override def authenticate(secretKey: String): Unit = {
    val res = redisClient.auth(secretKey)

    //TODO return as value. Try / Either or others..
    if (!res) {
      throw new Exception(f"Could not authenticate redis")
    }
  }

  override def get(key: String): Option[String] = {
    redisClient.get[String](key)
  }

  override def set(key: String, value: Any): Boolean =
    redisClient.set(key, value)
}

object RedisOpsSingleThread {
  def apply(host: String, port: Int): RedisOpsSingleThread = {
    val redisClient: RedisClient =
      new RedisClient(host, port)
    RedisOpsSingleThread(redisClient)
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the `RedisClient`

Solution:

- Phantom Types!
  - Build the "states" as `sealed trait` inside the `RedisOpsT` companion object
  - Add a type constraint on our trait
  - Align in all implementations

```scala
case class RedisOpsSingleThread[S <: State](redisClient: RedisClient)
    extends RedisOpsT[S] {
  override def authenticate(secretKey: String): RedisOpsT[State.Authenticated] = {
  redisClient
    .auth(secretKey)
    .fold(
      //return new RedisOps with Authenticated state
      RedisOpsSingleThread[State.Authenticated](redisClient),
      throw new Exception(f"Could not authenticate redis")
    )
}

  override def get[S <: State.Authenticated](key: String): Option[String] = {
    redisClient.get[String](key)
  }

  override def set[S <: State.Authenticated](key: String, value: Any): Boolean =
    redisClient.set(key, value)
}

object RedisOpsSingleThread {
  def apply(host: String, port: Int):
    RedisOpsSingleThread[State.UnAuthenticated] = {
      val redisClient: RedisClient =
        new RedisClient(host, port)
      RedisOpsSingleThread[State.UnAuthenticated](redisClient)
  }
}
```

TODO:

- "Embed" somehow the "authentication" state in the RedisClient

---

Solution:

- Phantom Types!
  - Build the "states" as sealed trait inside the RedisOpsT companion object
  - Add a type constraint on our trait
  - Align in all implementations

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
```

```scala
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT = RedisOpsSingleThread("host", 9379)
    redisClient.authenticate("secret")
    println("cost serving http server has started..")

    startHttpServer()
  }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequest)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What if we do a redisClient.get on an UnAuthenticated client?

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

What if we do a redisClient.get on an UnAuthenticated client?

```
inferred type arguments
[String,com.appsflyer.cost.common.redis.RedisOpsT.State.UnAuthenticated]
        do not conform to method get's type parameter bounds [
        com.appsflyer.cost.common.redis.RedisOpsT.State.UnAuthenticated <:
        com.appsflyer.cost.common.redis.RedisOpsT.State.Authenticated]
  redisClient.get("someKey")
```

Build will fail!

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- ✓ Allow testability to methods using RedisClient
- “Embed” somehow the “authentication” state in the RedisClient
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- ✓ Allow testability to methods using RedisClient
- ✓ “Embed” somehow the “authentication” state in the RedisClient
- ✓ Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

TODO:

- Better “key generation” than string interpolation!

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class

```scala
trait RedisOpsT[S <: RedisOpsT.State] {
  def authenticate(secret: String): RedisOpsT[State.Authenticated]
  def get[S <: State.Authenticated](key: String): Option[String]
  def set[S <: State.Authenticated](key: String, value: Any): Boolean
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class

```scala
trait RedisOpsT[S <: RedisOpsT.State] {
  def authenticate(secret: String): RedisOpsT[State.Authenticated]
  def get[S <: State.Authenticated](key: String): Option[String]
  def set[S <: State.Authenticated](key: String, value: Any): Boolean
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param

```scala
trait RedisOpsT[K: RedisKeyGenerator, S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K): Option[String]

  def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K, value: Any): Boolean

}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param

```scala
trait RedisOpsT[K: RedisKeyGenerator, S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K): Option[String]

  def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K, value: Any): Boolean

}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param

```scala
trait RedisOpsT[K: RedisKeyGenerator, S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K): Option[String]

  def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K, value: Any): Boolean

}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param

```scala
trait RedisOpsT[K: RedisKeyGenerator, S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K): Option[String]

  def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K, value: Any): Boolean

}

case class RedisOpsSingleThread[S <: State](redisClient: RedisClient)
    extends RedisOpsT[S] {
  override def authenticate(secretKey: String):
      RedisOpsT[State.Authenticated] = {
  redisClient
    .auth(secretKey)
    .fold(
      //return new RedisOps with Authenticated state
      RedisOpsSingleThread[State.Authenticated](redisClient),
      throw new Exception(f"Could not authenticate redis"))
}

  override def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: String): Option[String] = {
    redisClient.get[String](RedisKeyGenerator[K].toKey(key))
  }

  override def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: String, value: Any): Boolean =
    redisClient.set(RedisKeyGenerator[K].toKey(key), value)
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes

```scala
trait RedisOpsT[K: RedisKeyGenerator, S <: RedisOpsT.State] {

  def authenticate(secret: String): RedisOpsT[State.Authenticated]

  def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K): Option[String]

  def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: K, value: Any): Boolean

}

case class RedisOpsSingleThread[S <: State](redisClient: RedisClient)
    extends RedisOpsT[S] {
  override def authenticate(secretKey: String):
      RedisOpsT[State.Authenticated] = {
  redisClient
    .auth(secretKey)
    .fold(
      //return new RedisOps with Authenticated state
      RedisOpsSingleThread[State.Authenticated](redisClient),
      throw new Exception(f"Could not authenticate redis"))
}

  override def get[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: String): Option[String] = {
    redisClient.get[String](RedisKeyGenerator[K].toKey(key))
  }

  override def set[K: RedisKeyGenerator, S <: State.Authenticated]
      (key: String, value: Any): Boolean =
    redisClient.set(RedisKeyGenerator[K].toKey(key), value)
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!

```scala
trait RedisKeyGenerator[K] {
  def toKey(k: K): String
}

object RedisKeyGenerator {
  def apply[K](implicit r: RedisKey[K]):
    RedisKeyGenerator[K] = r

  def redisKey[K: RedisKey](k: K): String =
    RedisKeyGenerator[K].toKey(k)

  object RedisKeyGeneratorOps {
    implicit val stringRedisKey: RedisKeyGenerator[String] =
    (k: String) ⇒ k
  }
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!

```scala
trait RedisKeyGenerator[K] {
  def toKey(k: K): String
}

object RedisKeyGenerator {
  def apply[K](implicit r: RedisKey[K]):
    RedisKeyGenerator[K] = r

  def redisKey[K: RedisKey](k: K): String =
    RedisKeyGenerator[K].toKey(k)

  object RedisKeyGeneratorOps {
    implicit val stringRedisKey: RedisKeyGenerator[String] =
    (k: String) ⇒ k
  }
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

```scala
private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

```scala
private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = costRes,
    value = toJson(costRes))
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

```scala
sealed trait CostModel {
  val cost: Double
  val date: LocalDate
  val accountId: AccountId

  override def toString: String =
    """
      |cost: $cost
      |date: $date
      |accountId: $accountId
      |""".stripMargin
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

```scala
sealed trait CostModel {
  val cost: Double
  val date: LocalDate
  val accountId: AccountId

  override def toString: String =
    """
      |cost: $cost
      |date: $date
      |accountId: $accountId
      |""".stripMargin
}


object CostModel {

  implicit redisKeyGen: RedisKeyGenerator[CostModel] =
    (c: CostModel ⇒ s"${c.costType}-${c.accountId}-${c.date}"")
}
```

TODO:

- Better “key generation” than string interpolation!

Solution:

- Type class
  - Add a type to “key” param
  - Align all extended classes
  - Build the type class!
  - Align in code

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = costRes,
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequest)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = costRes,
    value = toJson(costRes))
```

TODO:

- ✓ Allow testability to methods using RedisClient
- ✓ “Embed” somehow the “authentication” state in the RedisClient
- ✓ Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequest)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = costRes,
    value = toJson(costRes))
```

TODO:

- Allow testability to methods using RedisClient
- “Embed” somehow the “authentication” state in the RedisClient
- Methods to receive already authenticated client!
- Better “key generation” than string interpolation!

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
   implicit val redisClient: RedisClient = new RedisClient("host", 6379)
   authenticateRedis(redisClient, "secret")
   println("cost serving http server has started..")

   startHttpServer()
 }

def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisClient):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisClient
): IO[CostModel] = {
  getCostFromCache(accountId, date, costType).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}

private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisClient
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisClient) =
  redisClient.set(
    key = s"${costRes.costType}-${costRes.accountId}-${costRes.date}",
    value = toJson(costRes))
```

```scala
def main(args: Array[String]): Unit = {
    implicit val redisClient: RedisOpsT[State.UnAuthenticated] =
      RedisOpsSingleThread("host", 9379)

    println("cost serving http server has started..")

    startHttpServer()
  }
def handleCostRequest(request: CostHttpRequst)
                     (implicit redisClient: RedisOpsT[State.UnAuthenticated]):
  IO[HttpResponse] = {
  for {
    costResult ← getCost(request.accountId,
                         request.date,
                         request.costModel)
    _ ← logger.info(s"received: $costResult")
    _ ← setToCache(costResult)
  } yield buildRespons(costResult)
}

private def getCost(accountId: AccountId,
                    date: LocalDate,
                    costType: CostType)
  (implicit redisClient: RedisOpsT[State.UnAuthenticated]
): IO[CostModel] = {
  val authenticatedRedisClient = redisClient.authenticate("secret")
  getCostFromCache(accountId, date, costType)
    (authenticatedRedisClient).map(
    _.getOrElse(getCostFromDB(accountId, date, costType))
  )
}
private def getCostFromCache(id: AccountId,
                             date: LocalDate,
                             costType: CostType)
  (implicit redisClient: RedisOpsT[State.Authenticated]
): IO[Option[CostModel]] = IO {
  redisClient.get(s"$costType-$id-$date") }

private def setToCache(costRes: CostModel)
    (implicit redisClient: RedisOpsT[State.Authenticated]) =
  redisClient.set(
    key = costRes,
    value = toJson(costRes))
```

- Thank you -