How to build an Event-Sourcing system Ho using Akka with EKS

ScalaMatsuri 2019

Junichi Kato(@j5ik2o)

Chatwork

Who am I

- Chatwork Tech-Lead
- github/j5ik2o
    - scala-ddd-base
    - scala-ddd-base-akka-http.g8
    - reactive-redis
    - reactive-memcached
- 翻訳レビュー
    - エリックエヴァンスのドメイン駆動設計
    - Akka実践バイブル

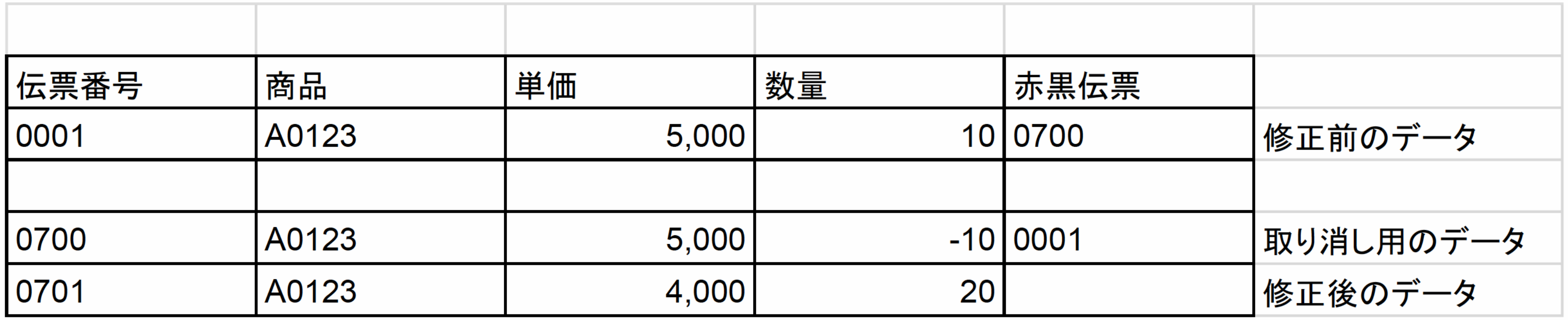
Agenda

1. Event Sourcing with Akka
2. Deployment to EKS

Akka with Event Sourcing

Event Sourcing

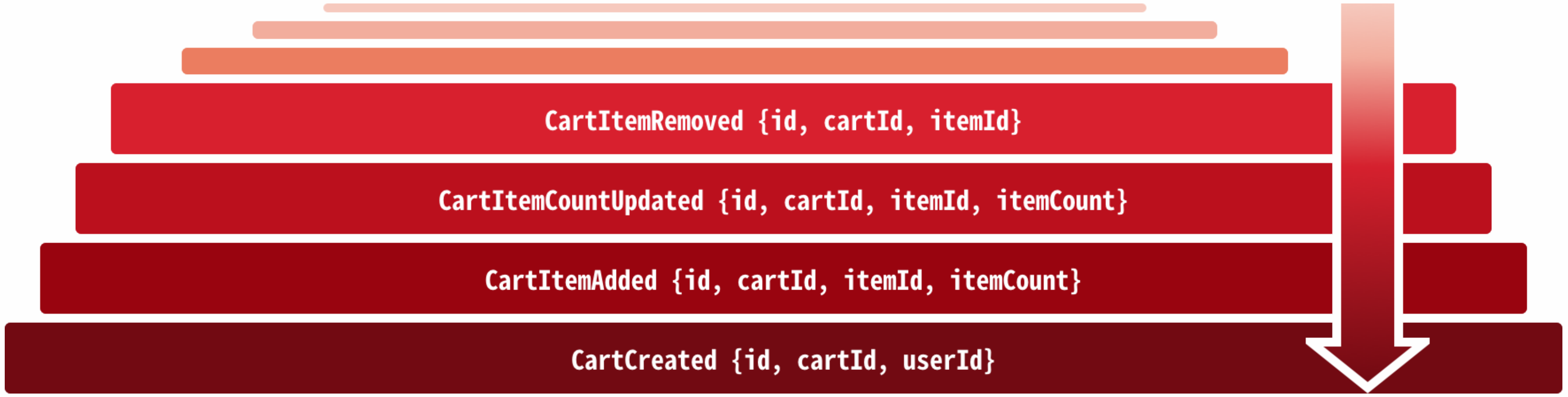
- The latest state is derived by the events
- For example, transactions such as the e-commerce are sourced on events. This is nothing special.
- An event sequence represents an immutable history.
    - The transaction makes the following unique corrections. Events are never modified or deleted.
    - The order #0001 is canceled at the #0700, and the corrected data is registered at the slip #0701.



伝票番号	商品	単価	数量	赤黒伝票	
0001	A0123	5,000	10	0700	修正前のデータ
0700	A0123	5,000	-10	0001	取り消し用のデータ
0701	A0123	4,000	20		修正後のデータ

Domain Events

- Events that occurred in the past
- Domain Events are events that domain experts is interested in
- Generally, Domain Events is expressed as a verb in past tense
  - CustomerRelocated
  - CargoShipped
- Events and commands are similar, but different languages are handled by humans
  - Command may be rejected
  - Indicates that the event has already occurred

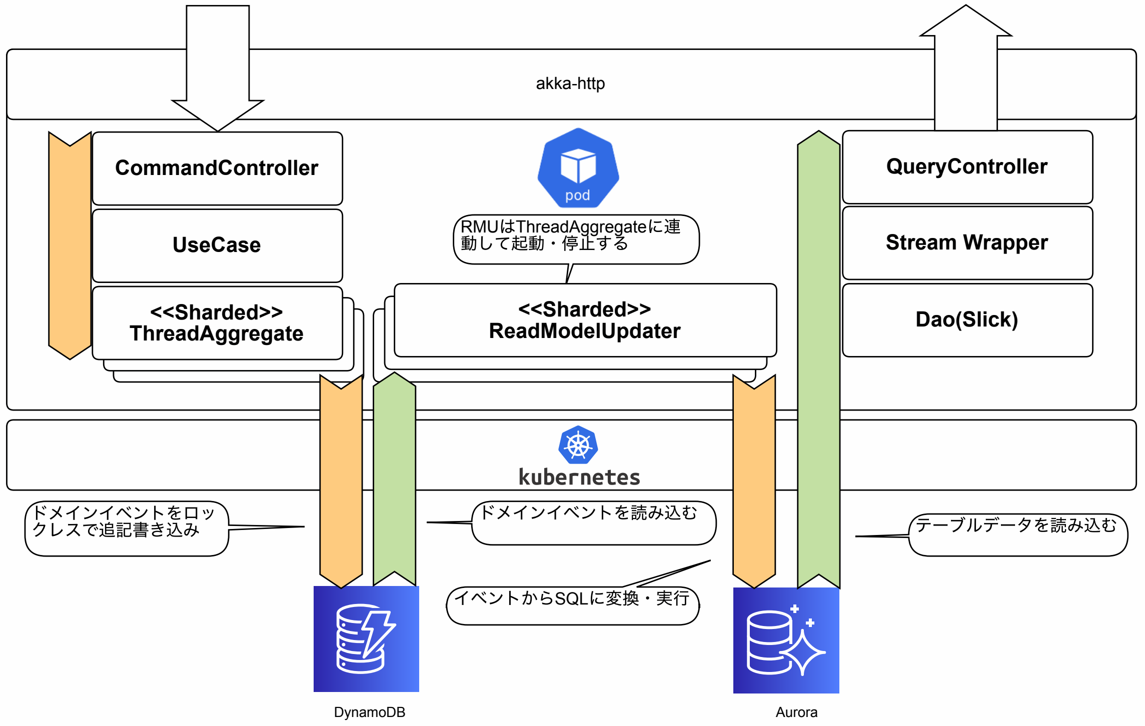

CartItemRemoved {id, cartId, itemId}
CartItemCountUpdated {id, cartId, itemId, itemCount}
CartItemAdded {id, cartId, itemId, itemCount}
CartCreated {id, cartId, userId}

**Consider thread-weaver
as an example of a simple chat application.**

System requirements

- API server accepts commands and queries from API clients
- Create a thread to start the chat
- Only members can post to threads
- Only text messages posted to threads
- Omit authentication and authorization for convenience

System Configuration

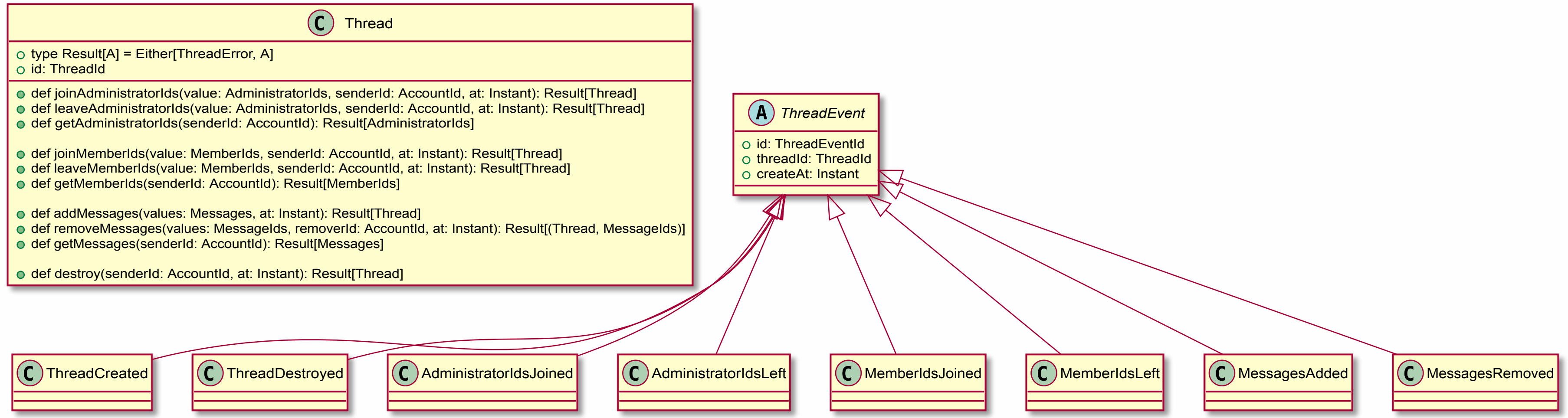

akka-http
CommandController
UseCase
<<Sharded>> ThreadAggregate
pod
RMUはThreadAggregateに連動して起動・停止する
<<Sharded>> ReadModelUpdater
QueryController
Stream Wrapper
Dao(Slick)
kubernetes
ドメインイベントをロックレスで追記書き込み
ドメインイベントを読み込む
イベントからSQLに変換・実行
テーブルデータを読み込む
DynamoDB
Aurora

- Split the application into the command stack and the query stack
- The command is sent to (clustered sharding) aggregate actor
- The aggregate actor stores(appends) domain events in storage when it accepts a command
- RMU(cluster sharding) starts up in conjunction with the aggregation actor and reads the domain events for the appropriate aggregate ID immediately after startup, executes the SQL, and creates the Read-Model
- Query using DAO to load and return the lead model
- Deploy the api-server as a kubernetes pod

Command stack side

Domain Objects

- Account
  - Account information identifying the user of the system
- Thread
  - Indicates a place to exchange Messages
- Message
  - A hearsay written in some language
- Administrator
  - Administrator of the Thread
- Member
  - Users of the Thread



```
Thread
type Result[A] = Either[ThreadError, A]
id: ThreadId
def joinAdministratorIds(value: AdministratorIds, senderId: AccountId, at: Instant): Result[Thread]
def leaveAdministratorIds(value: AdministratorIds, senderId: AccountId, at: Instant): Result[Thread]
def getAdministratorIds(senderId: AccountId): Result[AdministratorIds]

def joinMemberIds(value: MemberIds, senderId: AccountId, at: Instant): Result[Thread]
def leaveMemberIds(value: MemberIds, senderId: AccountId, at: Instant): Result[Thread]
def getMemberIds(senderId: AccountId): Result[MemberIds]

def addMessages(values: Messages, at: Instant): Result[Thread]
def removeMessages(values: MessageIds, removerId: AccountId, at: Instant): Result[(Thread, MessageIds)]
def getMessages(senderId: AccountId): Result[Messages]

def destroy(senderId: AccountId, at: Instant): Result[Thread]
```

```
ThreadEvent
id: ThreadEventId
threadId: ThreadId
createAt: Instant
```

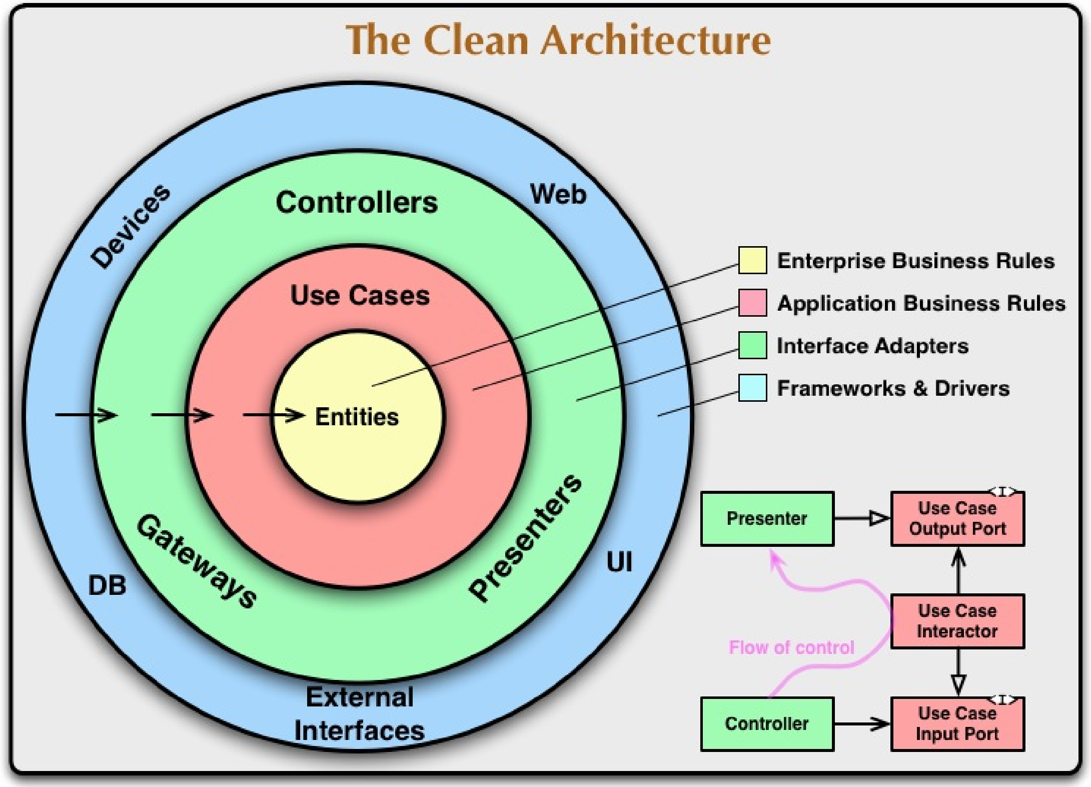
Commands/Domain Events

ThreadEvent sub types

- Create/Destroy Thread
  - ThreadCreated
  - ThreadDestroyed
- Join/Leave AdministratorIds
  - AdministratorIdsJoined
  - AdministratorIdsLeft
- Join/Leave MemberIds
  - MemberIdsJoined
  - MemberIdsLeft
- Add/Remove Messages
  - MessagesAdded
  - MessagesRemoved

Layered architecture

- Clean Architecture
- Common
  - interface-adaptors
  - infrastructure
- Command side
  - use-cases
  - domain
- Query side
  - data access streams
  - data access objects



Projects structure

Domain objects with actors

- Actors that fulfill all the functions are undesirable
- Follow object-oriented principles to build a hierarchy of actors with a single responsibility

Thread

```scala
trait Thread {

  def isAdministratorId(accountId: AccountId): Boolean
  def isMemberId(accountId: AccountId): Boolean

  def joinAdministratorIds(value: AdministratorIds, senderId: AccountId, at: Instant): Result[Thread]
  def leaveAdministratorIds(value: AdministratorIds, senderId: AccountId, at: Instant): Result[Thread]
  def getAdministratorIds(senderId: AccountId): Result[AdministratorIds]

  def joinMemberIds(value: MemberIds, senderId: AccountId, at: Instant): Result[Thread]
  def leaveMemberIds(value: MemberIds, senderId: AccountId, at: Instant): Result[Thread]
  def getMemberIds(senderId: AccountId): Result[MemberIds]

  def addMessages(values: Messages, at: Instant): Result[Thread]
  def removeMessages(values: MessageIds, removerId: AccountId, at: Instant): Result[(Thread, MessageIds)]
  def getMessages(senderId: AccountId): Result[Messages]

  def destroy(senderId: AccountId, at: Instant): Result[Thread]
}
```

ThreadAggregate

```scala
class ThreadAggregate(id: ThreadId,
  subscribers: Seq[ActorRef]) extends Actor {
  // add messages handler
  private def commandAddMessages(thread: Thread): Receive = {
    case AddMessages(requestId, threadId,
      messages, createAt, reply) if threadId == id =>
      thread.addMessages(messages, createAt) match {
        case Left(exception) =>
          if (reply)
            sender() ! AddMessagesFailed(ULID(), requestId,
              threadId, exception.getMessage, createAt)
        case Right(newThread) =>
          if (reply)
            sender() ! AddMessagesSucceeded(ULID(), requestId,
              threadId, messages.toMessageIds, createAt)
          context.become(onCreated(newThread))
      }
  }

  override def receive: Receive = { /*...*/ }
}
```

- Actors that support transactional integrity
- The boundary of the data update is the same as the boundary the aggregates has.
- For example, when an actor receives the CreateThead command, a Thread state is generated internally
- Then Messages are also added to the Thread when the AddMessages command is receives
- If the other commands defined in the protocol are received by the Actor, the Actor will have corresponding side effects.

ThreadAggreateSpec

```scala
val threadId        = ThreadId()
val threadRef       = newThreadRef(threadId)
val now             = Instant.now
val administratorId = AccountId()
val title           = ThreadTitle("test")

threadRef ! CreateThread(ULID(), threadId, administratorId,
  None, title, None, AdministratorIds(administratorId),
  MemberIds.empty, now, reply = false)

val messages = Messages(TextMessage(MessageId(), None,
  ToAccountIds.empty, Text("ABC"), memberId, now, now))
threadRef ! AddMessages(ULID(), threadId, messages,
  now, reply = true)

expectMsgType[AddMessagesResponse] match {
 case f: AddMessagesFailed =>
   fail(f.message)
 case s: AddMessagesSucceeded =>
   s.threadId shouldBe threadId
   s.createAt shouldBe now
}
```

- Verify that add messages and create a thread by using Test Kit

PersistentThreadAggregate(1/2)

```
object PersistentThreadAggregate {
  def props(id: ThreadId, subscribers: Seq[ActorRef]): Props =
      ...
}

class PersistentThreadAggregate(id: ThreadId,
  subscribers: Seq[ActorRef],
  propsF: (ThreadId, Seq[ActorRef]) => Props)
    extends PersistentActor with ActorLogging {

  override def supervisorStrategy: SupervisorStrategy =
    OneForOneStrategy() { case _: Throwable => Stop }

  private val childRef =
    context.actorOf(propsF(id, subscribers),
      name = ThreadAggregate.name(id))
  context.watch(childRef)

  // persistenceIdえ is the partition keでもスレッドIDに対応するf persi

  override def receiveRecover: Receive = {
    case e: ThreadCommonProtocol.Event with ToCommandRequest =>
      childRef ! e.toCommandRequest
    case RecoveryCompleted =>
      log.debug("recovery completed")
  }
```

- Actors that add the persistence function to ThreadAggregate
- Domain behavior is provided by child actors
- The recover process sends commands generated from events to child actors.

PersistentThreadAggregate(2/2)

```
  override def receiveCommand: Receive = {
    case Terminated(c) if c == childRef =>
      context.stop(self)
    case m: CommandRequest with ToEvent =>
      childRef ! m
      context.become(sending(sender(), m.toEvent))
    case m =>
      childRef forward m
  }

  private def sending(replyTo: ActorRef,
    event: ThreadCommonProtocol.Event): Receive = {
    case s: CommandSuccessResponse => persist(event) { _ =>
        replyTo ! s
        unstashAll()
        context.unbecome()
      }
    case f: CommandFailureResponse =>
      replyTo ! f
      unstashAll()
      context.unbecome()
    case _ =>
      stash()
  }
}
```

- Delegate to child actors when receiving commands. Persists only on success
- message processing is suspended until a command response is returned

PersitentThreadAggregateSpec

```
// Create id = 1 of Thread actor
threadRef1 ! CreateThread(ULID(), threadId, administratorId, None, title, None,
  AdministratorIds(administratorId), MemberIds.empty, now, reply = false)
val messages = Messages(TextMessage(MessageId(), None,
  ToAccountIds.empty, Text("ABC"), memberId, now, now))
threadRef1 ! AddMessages(ULID(), threadId, messages, now, reply = false)

//Stop id = 1 of Thread actor
killActors(threadRef)

// Recover id = 1 of Thread actor
val threadRef2 = system.actorOf(PersistentThreadAggregate.props(threadId, Seq.empty))

// Check if it is in the previous state
threadRef2 ! GetMessages(ULID(), threadId, memberId, now)
expectMsgType[GetMessagesResponse] match {
  case f: GetMessagesFailed =>
    fail(f.message)
  case s: GetMessagesSucceeded =>
    s.threadId shouldBe threadId
    s.createAt shouldBe now
    s.messages shouldBe messages
}
```

- a test that intentionally stops and restarts the persistence actor
- Replayed state after reboot

FYI: akka-persistence plugin

- プラグインを変えることで様々なデータベースに対応できる
  - Akka Persistence journal and snapshot plugins
- デフォルトはLevelDBに対応している
- AWSでのお勧めはDynamoDB。本家と私が作っているプラグインがあるが、おすすめは私のほうです
  - https://github.com/j5ik2o/akka-persistence-dynamodb

```
libraryDependencies ++= Seq(
  // ...
  "com.typesafe.akka" %% "akka-persistence" % akkaVersion,
  "org.fusesource.leveldbjni" % "leveldbjni-all" % "1.8" % Tes
  "org.iq80.leveldb" % "leveldb" % "0.9" % Test,
  "com.github.j5ik2o" %% "akka-persistence-dynamodb" % "1.0.2",
  // ...
)
```

```
akka {
  persistence {
    journal {
      plugin = dynamo-db-journal
    }
    snapshot-store {
      plugin = dynamo-db-snapshot
    }
  }
}
```

ThreadAggregates(Message Broker)

```scala
class ThreadAggregates(subscribers: Seq[ActorRef],
    propsF: (ThreadId, Seq[ActorRef]) => Props)
    extends Actor
    with ActorLogging
    with ChildActorLookup {
  override type ID             = ThreadId
  override type CommandRequest = ThreadProtocol.CommandMessage

  override def receive: Receive = forwardToActor

  override protected def childName(childId: ThreadId): String =
    childId.value.asString
  override protected def childProps(childId: ThreadId): Props =
    propsF(childId, subscribers)
  override protected def toChildId(commandRequest: CommandMessage): ThreadId =
    commandRequest.threadId
}
```

- The message broker that bundles multiple ThreadAggregates as child actors
- Most of the logic is in ChildActorLookup
- Resolve the actor name from ThreadId in the command message, and transfer the message to the corresponding child actor. If there is no child actor, generate an actor and then forward the message to the actor

ChildActorLookup

```scala
trait ChildActorLookup extends ActorLogging { this: Actor =>
  implicit def context: ActorContext
  type ID
  type CommandRequest

  protected def childName(childId: ID): String
  protected def childProps(childId: ID): Props
  protected def toChildId(commandRequest: CommandRequest): ID

  protected def forwardToActor: Actor.Receive = {
    case _cmd =>
      val cmd = _cmd.asInstanceOf[CommandRequest]
      context
        .child(childName(toChildId(cmd)))
        .fold(createAndForward(cmd, toChildId(cmd)))(forwardCommand(cmd))
  }

  protected def forwardCommand(cmd: CommandRequest)(childRef: ActorRef): Unit =
    childRef forward cmd

  protected def createAndForward(cmd: CommandRequest, childId: ID): Unit =
    createActor(childId) forward cmd

  protected def createActor(childId: ID): ActorRef =
    context.actorOf(childProps(childId), childName(childId))
}
```

- Create a child actor if none exists and forward the message
- forward the message to its child actors, if any

ShardedThreadAggregates (1/2)

```scala
object ShardedThreadAggregates {

  def props(subscribers: Seq[ActorRef],
    propsF: (ThreadId, Seq[ActorRef]) => Props): Props =
    Props(new ShardedThreadAggregates(subscribers, propsF))

  def name(id: ThreadId): String = id.value.asString

  val shardName = "threads"

  case object StopThread

  // function to extract an entity id
  val extractEntityId: ShardRegion.ExtractEntityId = {
    case cmd: CommandRequest => (cmd.threadId.value.asString, cmd)
  }

  // function to extract a shard id
  val extractShardId: ShardRegion.ExtractShardId = {
    case cmd: CommandRequest =>
      val mostSignificantBits  = cmd.threadId
        .value.mostSignificantBits  % 12
      val leastSignificantBits = cmd.threadId
        .value.leastSignificantBits % 12
      s"$mostSignificantBits:$leastSignificantBits"
  }

}
```

- Allow ThreadAggregates to be distributed across a cluster
- extractEntityId is the function to extract an entity id
- extractShardId is the function to extract a shard id

ShardedThreadAggregates (2/2)

```
class ShardedThreadAggregates(subscribers: Seq[ActorRef],
  propsF: (ThreadId, Seq[ActorRef]) => Props)
    extends ThreadAggregates(subscribers, propsF) {
  context.setReceiveTimeout(
    Settings(context.system).passivateTimeout)

  override def unhandled(message: Any): Unit = message match {
    case ReceiveTimeout =>
      log.debug("ReceiveTimeout")
      context.parent ! Passivate(stopMessage = StopThread)
    case StopThread =>
      log.debug("StopWallet")
      context.stop(self)
  }
}
```

- Inherit ThreadAggregates
- Then add an implementation to passivate ShardedThreadAggregates when occurred ReceiveTimeout

ShardedThreadAggregatesRegion

```scala
object ShardedThreadAggregatesRegion {

  def startClusterSharding(subscribers: Seq[ActorRef])
    (implicit system: ActorSystem): ActorRef =
    ClusterSharding(system).start(
      ShardedThreadAggregates.shardName,
      ShardedThreadAggregates.props(subscribers,
        PersistentThreadAggregate.props),
      ClusterShardingSettings(system),
      ShardedThreadAggregates.extractEntityId,
      ShardedThreadAggregates.extractShardId
    )

  def shardRegion(implicit system: ActorSystem): ActorRef =
    ClusterSharding(system)
      .shardRegion(ShardedThreadAggregates.shardName)
}
```

- The startClusterSharing method will start ClusterSharing with the specified settings
- The shardRegion method gets the ActorRef to the started ShardRegion.

MultiJVM Testing

```
"setup shared journal" in {
  Persistence(system)
  runOn(controller) { system.actorOf(Props[SharedLeveldbStore], "store") }
  enterBarrier("persistence-started")
  runOn(node1, node2) {
    system.actorSelection(node(controller) / "user" / "store") ! Identify(None)
    val sharedStore = expectMsgType[ActorIdentity].ref.get
    SharedLeveldbJournal.setStore(sharedStore, system)
  }
  enterBarrier("setup shared journal")
}
"join cluster" in within(15 seconds) {
  join(node1, node1) { ShardedThreadAggregatesRegion.startClusterSharding(Seq.empty) }
  join(node2, node1) { ShardedThreadAggregatesRegion.startClusterSharding(Seq.empty) }
  enterBarrier("join cluster")
}
"createThread" in { runOn(node1) {
    val accountId = AccountId(); val threadId  = ThreadId(); val title = ThreadTitle("test")
    val threadRef = ShardedThreadAggregatesRegion.shardRegion
    threadRef ! CreateThread(ULID(), threadId, accountId, None, title, None, AdministratorIds(accountId),
      MemberIds.empty, Instant.now, reply = true)
    expectMsgType[CreateThreadSucceeded](file:///Users/j5ik2o/Sources/thread-weaver/slide/10 seconds).threadId shouldBe threadId
  }
  enterBarrier("create thread")
}
```

cluster-sharding with persistence

- Actors with state in on-memory are distributed across the cluster
- Domain events that occur are saved in partitioned storage by aggregate ID

CreateThreadUseCaseUntypeImpl

```scala
class CreateThreadUseCaseUntypeImpl(
    threadAggregates: ThreadActorRefOfCommandUntypeRef, parallelism: Int = 1, timeout: Timeout = 3 seconds
)(implicit system: ActorSystem) extends CreateThreadUseCase {
  override def execute: Flow[UCreateThread, UCreateThreadResponse, NotUsed] =
    Flow[UCreateThread].mapAsync(parallelism) { request =>
      implicit val to: Timeout                    = timeout
      implicit val scheduler: Scheduler           = system.scheduler
      implicit val ec: ExecutionContextExecutor = system.dispatcher
      (threadAggregates ? CreateThread(
        ULID(), request.threadId, request.creatorId, None, request.title, request.remarks,
        request.administratorIds, request.memberIds, request.createAt, reply = true
      )).mapTo[CreateThreadResponse].map {
        case s: CreateThreadSucceeded =>
          UCreateThreadSucceeded(s.id, s.requestId, s.threadId, s.createAt)
        case f: CreateThreadFailed =>
          UCreateThreadFailed(f.id, f.requestId, f.threadId, f.message, f.createAt)
      }
    }
}
```

ThreadCommandControllerImpl

```scala
trait ThreadCommandControllerImpl
  extends ThreadCommandController
  with ThreadValidateDirectives {
  private val createThreadUseCase = bind[CreateThreadUseCase]
  private val createThreadPresenter = bind[CreateThreadPresenter]

  override private[controller] def createThread: Route =
    path("threads" / "create") {
      post {
        extractMaterializer { implicit mat =>
          entity(as[CreateThreadRequestJson]) { json =>
            validateJsonRequest(json).apply { commandRequest =>
              val responseFuture = Source.single(commandRequest)
                .via(createThreadUseCase.execute)
                .via(createThreadPresenter.response)
                .runWith(Sink.head)
              onSuccess(responseFuture) { response =>
                complete(response)
              }
            }
          }
        }
      }
    }
```

- Command side controller
- The thread creation root composes several directives and calls a use case
- The request JSON returns a command if validation passes. Pass the command to the use-case and execute it
- The presenter will convert the use-case result to Response JSON

Read Model Updater side

FYI: akka-typed

- メッセージハンドラで受け取るメッセージ型はAnyだったが、型を指定できるようになった
- 基本的に互換性がないので、覚えることが多い。今のうちになれておこう

```scala
object PingPong extends App {

  trait Message
  case class Ping(reply: ActorRef[Message]) extends Message
  case object Pong                          extends Message

  def receiver: Behavior[Message] =
    Behaviors.setup[Message] { ctx =>
      Behaviors.receiveMessagePartial[Message] {
        case Ping(replyTo) =>
          ctx.log.info("ping")
          replyTo ! Pong
          Behaviors.same
      }
    }

  def main: Behavior[Message] = Behaviors.setup { ctx =>
    val receiverRef = ctx.spawn(receiver, name = "receiver")
    receiverRef ! Ping(ctx.self)
    Behaviors.receiveMessagePartial[Message] {
      case Pong =>
        ctx.log.info("pong")
        receiverRef ! Ping(ctx.self)
        Behaviors.same
    }
  }

  ActorSystem(main, "ping-pong")

}
```

Read Model Updater(1/3)

- 集約IDごとにRead Model Updater(RMU)を起動させる
- 1ノードで複数のRMUが起動できるようにシャーディングする
- RMUの起動と停止は集約アクターのイベントをきっかけにする。実際にはAggregateToRMUでメッセージ変換を行う

Read Model Updater(2/3)

```scala
private def projectionSource(sqlBatchSize: Long, threadId: ThreadId)
  (implicit ec: ExecutionContext): Source[Vector[Unit], NotUsed] = {
  Source
    .fromFuture(
      db.run(getSequenceNrAction(threadId)).map(_.getOrElse(0L))
    ).log("lastSequenceNr").flatMapConcat { lastSequenceNr =>
      readJournal
        .eventsByPersistenceId(threadId.value.asString,
          lastSequenceNr + 1, Long.MaxValue)
    }.log("ee").via(sqlActionFlow(threadId))
    .batch(sqlBatchSize, ArrayBuffer(_))(_ :+ _).mapAsync(1) { sqlActions =>
      db.run(DBIO.sequence(sqlActions.result.toVector))
    }.withAttributes(logLevels)
}
```

- RMUは終わらないストリーム処理を行います
- persistenceIdでもスレッドIDに対応する最新のシーケンス番号を取得します
- スレッドIDと最新のシーケンス番号以降のイベントをreadJournalから読み込みます
- sqlActionFlowではイベントをSQLに変換します
- 最後にSQLをまとめて実行します(今回は非正規はやっていません。問い合わせパターンに柔軟に対応するためです)

Read Model Updater(2/2)

```scala
class ThreadReadModelUpdater(
    val readJournal: ReadJournalType,
    val profile: JdbcProfile, val db: JdbcProfile#Backend#Database
) extends ThreadComponent with ThreadMessageComponent ... {
  import profile.api._

  def behavior(sqlBatchSize: Long = 10,
    backoffSettings: Option[BackoffSettings] = None): Behavior[CommandRequest] =
    Behaviors.setup[CommandRequest] { ctx =>
      Behaviors.receiveMessagePartial[CommandRequest] {
        case s: Start =>
          ctx.child(s.threadId.value.asString) match {
            case None =>
              ctx.spawn(
                projectionBehavior(sqlBatchSize, backoffSettings, s.threadId),
                name = s"RMU-${s.threadId.value.asString}"
              ) ! s
            case _ =>
              ctx.log.warning(
                "RMU already has started: threadId = {}", s.threadId.value.asString)
          }
          Behaviors.same
      }
    }
    // ...
}
```

- Read Model UpdaterはStartメッセージを受け取るとストリーム処理を開始します。
- ストリーム処理は子アクターのタスクとして実行されます

ShardedThreadReadModelUpdater(1/2)

```scala
class ShardedThreadReadModelUpdater(
    val readJournal: ReadJournalType,
    val profile: JdbcProfile,
    val db: JdbcProfile#Backend#Database
) {
  val TypeKey: EntityTypeKey[CommandRequest] = EntityTypeKey[CommandRequest](file:///Users/j5ik2o/Sources/thread-weaver/slide/"threads

  private def behavior(
      receiveTimeout: FiniteDuration, sqlBatchSize: Long = 10, backoffSettings: Option[BackoffSettings] = None
  ): EntityContext => Behavior[CommandRequest] = { entityContext =>
    Behaviors.setup[CommandRequest] { ctx =>
      val childRef = ctx.spawn(new ThreadReadModelUpdater(readJournal, profile, db).behavior(sqlBatchSize, backoffSettings),
        name = "threads-rmu")
      Behaviors.receiveMessagePartial {
        case Idle => entityContext.shard ! ClusterSharding.Passivate(ctx.self); Behaviors.same
        case Stop => Behaviors.stopped
        case Stop(_, _, _) => ctx.self ! Idle; Behaviors.same
        case msg => childRef ! msg; Behaviors.same
      }
    }
  }
```

ShardedThreadReadModelUpdater(2/2)

```scala
  def initEntityActor(
      clusterSharding: ClusterSharding,
      receiveTimeout: FiniteDuration
  ): ActorRef[ShardingEnvelope[CommandRequest]] =
    clusterSharding.init(
      Entity(typeKey = TypeKey, createBehavior = behavior(receiveTimeout)).withStopMessage(Stop)
    )
}
```

- ShardedThreadReadModelUpdaterProxy

```scala
class ShardedThreadReadModelUpdaterProxy(
    val readJournal: ReadJournalType, val profile: JdbcProfile, val db: JdbcProfile#Backend#Database
) {
  def behavior(clusterSharding: ClusterSharding, receiveTimeout: FiniteDuration): Behavior[CommandRequest] =
    Behaviors.setup[CommandRequest] { _ =>
      val actorRef =
        new ShardedThreadReadModelUpdater(readJournal, profile, db).initEntityActor(clusterSharding, receiveTimeout)
      Behaviors.receiveMessagePartial[CommandRequest] {
        case msg =>
          actorRef ! typed.ShardingEnvelope(msg.threadId.value.asString, msg)
          Behaviors.same
      }
    }
}
```

Interlocking of Aggreagte and RMU

```scala
object AggregateToRMU {

  def behavior(
      rmuRef: ActorRef[ThreadReadModelUpdaterProtocol.CommandRequest]
  ): Behavior[ThreadCommonProtocol.Message] =
    Behaviors.setup[ThreadCommonProtocol.Message] { ctx =>
      Behaviors.receiveMessagePartial[ThreadCommonProtocol.Message] {
        case s: Started =>
          ctx.log.debug(s"RMU ! $s")
          rmuRef ! ThreadReadModelUpdaterProtocol.Start(
            ULID(), s.threadId, Instant.now)
          Behaviors.same
        case s: Stopped =>
          ctx.log.debug(s"RMU ! $s")
          rmuRef ! ThreadReadModelUpdaterProtocol.Stop(
            ULID(), s.threadId, Instant.now)
          Behaviors.same
      }

    }

}
```

- この二つのアクターは責務が異なるので分離されていますが、起動と停止は連動します
- この方法はあまりよくありません。ノード故障でRMUだけが停止した場合、再度Startメッセージを受信しないとリカバリできないからです。ThreadAggregateから定期的にハートビードを受ける方法もありますが、RMUをThreadAggreagteの子アクターにする方法もあります。

RMU for improvement

- もうひとつの実装パターンは、RMUをPersistentThreadAggregateの子アクターにする方法です。
- RMUを子アクターとしてwatchできるようになるので、万が一RMUが停止しても再起動が可能です。
- ただしPersistentThreadAggregateがRMU責務を担うことになります。責務の重複？

Query stack side

ThreadQueryControllerImpl

```scala
trait ThreadQueryControllerImpl
  extends ThreadQueryController
  with ThreadValidateDirectives {
  private val threadDas: ThreadDas = bind[ThreadDas]
  // ...
  override private[controller] def getThread: Route =
    path("threads" / Segment) { threadIdString => get {
      // ...
      parameter('account_id) { accountValue =>
        validateAccountId(accountValue) { accountId =>
          onSuccess(threadDas.getThreadByIdSource(accountId, threadId)
            .via(threadPresenter.response)
            .runWith(Sink.headOption[ThreadJson]).map(identity)) {
              case None =>
                reject(NotFoundRejection("thread is not found", None))
              case Some(response) =>
                complete(GetThreadResponseJson(response))
          }
        }
      }
    // ...
    }
  // ...
}
```

- The query side uses a stream wrapped Dao object instead of a use case. — Same as command side except for this.

ThreadControllerSpec

```scala
val administratorId = ULID().asString
val entity = CreateThreadRequestJson(
    administratorId, None, "test",
    None, Seq(administratorId), Seq.empty,
    Instant.now.toEpochMilli
  ).toHttpEntity

Post(RouteNames.CreateThread, entity) ~>
  commandController.createThread ~> check {
  response.status shouldEqual StatusCodes.OK
  val responseJson = responseAs[CreateThreadResponseJson]
  responseJson.isSuccessful shouldBe true
  val threadId = responseJson.threadId.get

  eventually { // repeat util read
    Get(RouteNames.GetThread(threadId, administratorId)) ~>
      queryController.getThread ~> check {
      response.status shouldEqual StatusCodes.OK
      val responseJson = responseAs[GetThreadResponseJson]
      responseJson.isSuccessful shouldBe true
    }
  }
}
```

- a test where two controllers are connected.
- Verify threads are readable after they are created
- Works fine

Bootstrap

- Start AkkaManagement and ClusterBootstrap and start akka-http server

```scala
object Main extends App {
  // ...

  implicit val system: ActorSystem                       = ActorSystem("thread-weaver-api-server", config)
  implicit val materializer: ActorMaterializer           = ActorMaterializer()
  implicit val executionContext: ExecutionContextExecutor = system.dispatcher
  implicit val cluster                                   = Cluster(system)

  AkkaManagement(system).start()
  ClusterBootstrap(system).start()

  // ...

  val routes = session
      .build[Routes].root ~ /* ... */

  val bindingFuture = Http().bindAndHandle(routes, host, port).map { serverBinding =>
    system.log.info(s"Server online at ${serverBinding.localAddress}")
    serverBinding
  }

  // ...

}
```

FYI: Akka Cluster

- Cluster Specification
- Node
    - A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a hostname:port:uid tuple.
- Cluster
    - A set of nodes joined together through the membership service.
- leader
    - A single node in the cluster that acts as the leader. Managing cluster convergence and membership state transitions.
- Seed Nodes
    - The seed nodes are configured contact points for new nodes joining the cluster. When a new node is started it sends a message to all seed nodes and then sends a join command to the seed node that answers first.

FYI: Akka Management

- Akka Management is a suite of tools for operating Akka Clusters.
- modules
  - akka-management: HTTP management endpoints and health checks
  - akka-managment-cluster-http: Provides HTTP endpoints for cluster monitoring and management
  - akka-managment-cluster-bootstrap: Supports cluster bootstrapping by using akka-discovery
  - akka-discovery-kubernetes-api: Module for managing k8s pod as a cluster member

Example for akka.conf(1/2)

- Sample Configuration for Production

```
akka {
  cluster {
    seed-nodes = [] # seed-nodes are empty because managed by akka-management
    auto-down-unreachable-after = off
  }

  remote {
    log-remote-lifecycle-events = on
    netty.tcp {
      hostname = "127.0.0.1"
      hostname = ${?HOSTNAME}
      port = 2551
      port = ${?THREAD_WEAVER_REMOTE_PORT}
      bind-hostname = "0.0.0.0"
    }
  }
```

Example for akka.conf(2/2)

- Sample configuration for akka-management and akka-discovery
- Configuration to find nodes from k8s pod information

```
discovery {
  method = kubernetes-api
  method = ${?THREAD_WEAVER_DISCOVERY_METHOD}
  kubernetes-api {
    pod-namespace = "thread-weaver"
    pod-namespace = ${?THREAD_WEAVER_K8S_NAMESPACE}
    pod-label-selector = "app=thread-weaver-api-server"
    pod-label-selector = ${?THREAD_WEAVER_K8S_SELECTOR}
    pod-port-name = "management"
    pod-port-name = ${?THREAD_WEAVER_K8S_MANAGEMENT_PORT}
  }
}
```

```
  management {
    http {
      hostname = "127.0.0.1"
      hostname = ${?HOSTNAME}
      port = 8558
      port = ${?THREAD_WEAVER_MANAGEMENT_PORT}
      bind-hostname = 0.0.0.0
      bind-port = 8558
    }
    cluster.bootstrap {
      contact-point-discovery {
        discovery-method = kubernetes-api
      }
    }
    contract-point {
      fallback-port = 8558
    }
  }
}
```

Deployment to EKS

FYI: Kubernetes/EKSを学ぶ

- Kubernetes公式サイト
- Amazon EKS
- Amazon EKS Workshop

build.sbt for deployment

project/plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.10")

addSbtPlugin("com.mintbeans" % "sbt-ecr" % "0.14.1")
```

build.sbt

- Dockerの設定

```scala
lazy val dockerCommonSettings = Seq(
  dockerBaseImage := "adoptopenjdk/openjdk8:x86_64-alpine-jdk8u191-b12",
  maintainer in Docker := "Junichi Kato <j5ik2o@gmail.com>",
  dockerUpdateLatest := true,
  bashScriptExtraDefines ++= Seq(
    "addJava -Xms${JVM_HEAP_MIN:-1024m}",
    "addJava -Xmx${JVM_HEAP_MAX:-1024m}",
    "addJava -XX:MaxMetaspaceSize=${JVM_META_MAX:-512M}",
    "addJava ${JVM_GC_OPTIONS:--XX:+UseG1GC}",
    "addJava -Dconfig.resource=${CONFIG_RESOURCE:-application.conf}",
    "addJava -Dakka.remote.startup-timeout=60s"
  )
)
```

build.sbt

- ECRに対する設定

```
val ecrSettings = Seq(
  region in Ecr := Region.getRegion(Regions.AP_NORTHEAST_1),
  repositoryName in Ecr := "j5ik2o/thread-weaver-api-server",
  repositoryTags in Ecr ++= Seq(version.value),
  localDockerImage in Ecr := "j5ik2o/" + (packageName in Docker).value + ":" + (version in Docker).value,
  push in Ecr := ((push in Ecr) dependsOn (publishLocal in Docker, login in Ecr)).value
)
```

build.sbt

```
val `api-server` = (project in file("api-server"))
  .enablePlugins(AshScriptPlugin, JavaAgent, EcrPlugin)
  .settings(baseSettings)
  .settings(dockerCommonSettings)
  .settings(ecrSettings)
  .settings(
    name := "thread-weaver-api-server",
    dockerEntrypoint := Seq("/opt/docker/bin/thread-weaver-api-server"),
    dockerUsername := Some("j5ik2o"),
  // ...
    libraryDependencies ++= Seq(
      "com.github.scopt" %% "scopt" % "4.0.0-RC2",
      "net.logstash.logback" % "logstash-logback-encoder" % "4.11" excludeAll (/**/),
      "com.lightbend.akka.management" %% "akka-management" % akkaManagementVersion,
      "com.lightbend.akka.management" %% "akka-management-cluster-http" % akkaManagementVersion,
      "com.lightbend.akka.management" %% "akka-management-cluster-bootstrap" % akkaManagementVersion,
      "com.lightbend.akka.discovery" %% "akka-discovery-kubernetes-api" % akkaManagementVersion,
      "com.github.TanUkkii007" %% "akka-cluster-custom-downing" % "0.0.12",
      "com.github.everpeace" %% "healthchecks-core" % "0.4.0",
      "com.github.everpeace" %% "healthchecks-k8s-probes" % "0.4.0",
      "org.slf4j" % "jul-to-slf4j" % "1.7.26",
      "ch.qos.logback" % "logback-classic" % "1.2.3",
      "org.codehaus.janino" % "janino" % "3.0.6"
    )
```

Deployment to Local Cluster

Deployment to Local Cluster

- minikubeの起動
- helmの導入
- ネームスペースとサービスアカウントの作成
- DBをデプロイ
- スキーマ作成
- アプリケーション用イメージのビルド
- アプリケーション用イメージのデプロイ

```
$ minikube start --vmdriver virtualbox \
  --kubernetes-version v1.12.8 --cpus 6 --memory 5000 --disk-size 30g
$ helm init
$ kubectl create namespace thread-weaver
$ kubectl create serviceaccount thread-weaver
$ helm install ./mysql --namespace thread-weaver \
    -f ./mysql/environments/${ENV_NAME}-values.yaml
$ helm install ./dynamodb --namespace thread-weaver \
    -f ./dynamodb/environments/${ENV_NAME}-values.yaml
$ sbt -Dmysql.host="$(minikube ip)" -Dmysql.port=30306 \
  'migrate-mysql/run'
$ DYNAMODB_HOST="$(minikube ip)" DYNAMODB_PORT=32000 \
  sbt 'migrate-dynamodb/run'
$ eval $(minikube docker-env)
$ sbt api-server/docker:publishLocal
$ helm install ./thread-weaver-api-server \
  --namespace thread-weaver \
  -f ./thread-weaver-api-server/environments/${ENV_NAME}-values.yaml
```

Helm charts

FYI: Helm

- Helm
  - Helm is 'The package manager for Kubernetes'
- Charts
  - Helm uses a packaging format called charts
- CLI
  - `helm init`
    - initialize Helm on both client and server(tiller)
  - `helm package`
    - package a chart directory into a chart archive
  - `helm install`
    - install a chart archive
  - `helm upgrade`
    - upgrade a release
  - `helm rollback`
    - roll back a release to a previous revision

deployment.yaml(1/2)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "name" . }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ template "name" . }}
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: {{ template "name" . }}
    spec:
      containers:
      - image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        imagePullPolicy: {{.Values.image.pullPolicy}}
        name: {{ template "name" . }}
```

```
env:
  - name: AWS_REGION
    value: "ap-northeast-1"
  - name: HOSTNAME
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: status.podIP
  - name: ENV_NAME
    value: {{.Values.envName | quote}}
  - name: CONFIG_RESOURCE
    value: {{.Values.configResource | quote}}
  - name: JVM_HEAP_MIN
    value: {{.Values.jvmHeapMin | quote}}
  - name: JVM_HEAP_MAX
    value: {{.Values.jvmHeapMax | quote}}
  - name: JVM_META_MAX
    value: {{.Values.jvmMetaMax | quote}}
```

deployment.yaml(2/2)

```
          - name: THREAD_WEAVER_SLICK_URL
            value: {{.Values.db.url | quote}}
          - name: THREAD_WEAVER_SLICK_USER
            value: {{.Values.db.user | quote}}
          - name: THREAD_WEAVER_SLICK_PASSWORD
            valueFrom:
              secretKeyRef:
                name: thread-weaver-app-secrets
                key: mysql.password
          - name: THREAD_WEAVER_SLICK_MAX_POOL_SIZE
            value: {{.Values.db.maxPoolSize | quote}}
          - name: THREAD_WEAVER_SLICK_MIN_IDLE_SIZE
            value: {{.Values.db.minIdleSize | quote}}
        ports:
        - name: remoting
          containerPort: 2551
        - name: {{ .Values.service.name }}
          containerPort: {{ .Values.service.internalPort }}
        - name: management
          containerPort: 8558
```

```
        readinessProbe:
          tcpSocket:
            port: 18080
            initialDelaySeconds: 60
            periodSeconds: 30
        livenessProbe:
          tcpSocket:
            port: 18080
            initialDelaySeconds: 60
            periodSeconds: 30
```

- イメージの名前やタグ、環境変数、ポートなどコンテナが起動するための設定を記述する

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ template "name" . }}
  labels:
    app: {{ template "name" . }}
    chart: {{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  selector:
    app: {{ template "name" . }}
  type: {{ .Values.service.type }}
  ports:
    - protocol: TCP
      name: api
      port: 8080
      targetPort: api
    - protocol: TCP
      name: management
      port: 8558
      targetPort: management
```

- 外部からアクセス可能にするためにLoadBalancerタイプのServiceを構築します

rbac.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: thread-weaver-api-server
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: thread-weaver-api-server
subjects:
  - kind: User
    name: system:serviceaccount:thread-weaver:default
roleRef:
  kind: Role
  name: thread-weaver-api-server
  apiGroup: rbac.authorization.k8s.io
```

- akka-discoveryがノードを見つけられるようにRBACを設定します

Verification for Local Cluster

```
API_HOST=$(minikube ip)
API_PORT=$(kubectl get svc thread-weaver-api-server -n thread-weaver -ojsonpath="{.spec.ports[?(@.name==\"api\")].port}")

ACCOUNT_ID=01DB5QXD4NP0XQTV92K42B3XBF
ADMINISTRATOR_ID=01DB5QXD4NP0XQTV92K42B3XBF

THREAD_ID=$(curl -v -X POST "http://$API_HOST:$API_PORT/v1/threads/create" -H "accept: application/json" -H "Content-Type: applicatior
    -d "{\"accountId\":\"${ACCOUNT_ID}\",\"title\":\"string\",\"remarks\":\"string\",\"administratorIds\":[\"${ADMINISTRATOR_ID}\"],\"
echo "THREAD_ID=$THREAD_ID"
sleep 3
curl -v -X GET "http://$API_HOST:$API_PORT/v1/threads/${THREAD_ID}?account_id=${ACCOUNT_ID}" -H "accept: application/json"
```

Deployment to Production Cluster

Build Kubernetes Cluster

- EKSクラスター以外に必要なモノをすべて作る
    - subnet
    - security group
    - ineternet-gw
    - eip
    - nat-gw
    - route table
    - ecr
    - rds(aurora)
    - dynamodb(with shema)

```
$ terraform plan
$ terraform apply
```

Build Kubernetes Cluster

- EKSクラスタを構築

```
$ eksctl create cluster \
    --name ${CLUSTER_NAME} \
    --region ${AWS_REGION} \
      --nodes ${NODES} \
      --nodes-min ${NODES_MIN} \
      --nodes-max ${NODES_MAX} \
      --node-type ${INSTANCE_TYPE} \
      --full-ecr-access \
    --node-ami ${NODE_AMI} \
    --version ${K8S_VERSION} \
    --nodegroup-name ${NODE_GROUP_NAME} \
      --vpc-private-subnets=${SUBNET_PRIVATE1},${SUBNET_PRIVATE2},${SUBNET_PRIVATE3} \
      --vpc-public-subnets=${SUBNET_PUBLIC1},${SUBNET_PUBLIC2},${SUBNET_PUBLIC3}
```

- 初期設定(RBAC設定など)

```
tools/eks/helm $ kubectl apply -f ./rbac-config.yaml
$ helm init
$ kubectl create namespace thread-weaver
$ kubectl create serviceaccount thread-weaver
tools/deploy/eks $ kubectl apply -f secret.yaml
```

Build Kubernetes Cluster

- docker build & push to ecr

```
$ AWS_DEFUALT_PROFILE=xxxxx sbt api-server/ecr:push
```

- flyway migrate(should be implemented as k8s job)

```
$ docker run --rm -v $(pwd)/tools/flyway/src/test/resources/db-migration:/flyway/sql -v $(pwd):/flyway/conf boxfuse/flyway migrate
```

- verification

```
API_HOST=$(kubectl get svc thread-weaver-api-server -n thread-weaver -ojsonpath="{.status.loadBalancer.ingress[0].hostname}")
API_PORT=$(kubectl get svc thread-weaver-api-server -n thread-weaver -ojsonpath="{.spec.ports[?(@.name==\"api\")].port}")

ACCOUNT_ID=01DB5QXD4NP0XQTV92K42B3XBF
ADMINISTRATOR_ID=01DB5QXD4NP0XQTV92K42B3XBF

THREAD_ID=$(curl -v -X POST "http://$API_HOST:$API_PORT/v1/threads/create"
-H "accept: application/json" -H "Content-Type: application/json" \
-d "{\"accountId\":\"${ACCOUNT_ID}\", ... "memberIds\":[\"${ACCOUNT_ID}\"],\"createAt\":10000}" | jq -r .threadId)
echo "THREAD_ID=$THREAD_ID"
sleep 3
curl -v -X GET "http://$API_HOST:$API_PORT/v1/threads/${THREAD_ID}?account_id=${ACCOUNT_ID}" -H "accept: application/json"
```

本番運用のために考慮すべき観点

- Event Schema Evolution
  - Persistence - Schema Evolution
- Split-Brain Resolver
  - Split Brain Resolver
  - TanUkkii007/akka-cluster-custom-downing
- Distributed Tracing
  - kamon-io/Kamon
  - alevkhomich/akka-tracing

まとめ

- ドメインイベントは、ドメインの分析と実装の両方で使えるツール
- 集約を跨がる整合性の問題は難しいが、解決方法がないわけではない

一緒に働くエンジニアを募集しています！

http://corp.chatwork.com/ja/recruit/

Chatwork