Tour of Heroes

Going Event-Sourced with AWS

Leonid Korogodski <leonid.korogodski@gmail.com>, Chariot Solutions

Table of Contents

Introduction

This whitepaper is part of a series, in which three Chariot Solutions developers describe three different approaches to implementing a backend for the Tour of Heroes tutorial of the Angular frontend framework, with different AWS deployment strategies. The purpose of the series is not to find the "best" implementation for the Tour of Heroes specifically but to use it as an example application with which to illustrate different kinds of backends, as well as different kinds of AWS deployments.

In *this* paper, we will develop an event-sourced backend, following the CQRS principles (Command Query Responsibility Segregation). Granted, the solution presented here will be an overkill—like shooting birds with a cannon—but, again, our purpose is not to find the easiest or the most efficient implementation of the Tour of Heroes (which is a very simple application). Rather, it is to illustrate an approach that can be taken in developing other applications.

The complete code for this example can be found in our GitHub repositories: in Scala and in Java. However, we will arrive at its final shape and form in several stages, as follows:

1. We will begin by discussing the CQRS principle—in particular, the Event Sourcing approach and how it can be implemented with Akka Persistence actors—as well as the kinds of applications that could benefit from this.
2. We will use the Lagom framework, developed by Lightbend, to implement and test an event-sourced microservice backend for the Tour of Heroes that runs in the development mode locally, using Lagom's embedded Apache Cassandra server. Lagom microservices can be developed in Scala and/or Java. We will illustrate the development process step by step for Scala only, for conciseness and clarity. We will also comment on the Java version toward the end of this whitepaper.
3. We will show how to replace the embedded Cassandra server with Amazon Keyspaces, a new AWS offering that debuted in December 2019 (as AWS Managed Cassandra Service, in the preview mode) and became generally available in April 2020.
4. We will discover that, due to certain differences between Amazon Keyspaces and Apache Cassandra (some of them possibly temporary), Amazon Keyspaces cannot yet be used as the read side of a Lagom application.
5. We will use this as a teaching moment to show how to modify the application to use different databases for the write side and the read side. Specifically, we will switch to a PostgreSQL database for the read side, while continuing to have Amazon Keyspaces as the write side.
6. We will show how to deploy the application to an AWS Kubernetes cluster, with AWS Aurora Serverless (in the PostgreSQL mode) as the read side. In the process, we will troubleshoot a few problems that may arise along the way.
7. We will use AWS Secrets Manager to protect the Amazon Keyspaces and AWS Aurora database credentials.
8. We will upload the static frontend assets to an Amazon S3 bucket and create an AWS CloudFront distribution to serve the static assets from the S3 bucket while forwarding server requests to an AWS load balancer created for the AWS Kubernetes cluster described in the previous step.
9. We will write AWS CLI scripts to deploy and upgrade the microservice. While there are other

ways to automate AWS deployment (AWS CloudFormation, Terraform, AWS CDK, etc.), some of which will also be described in this series, AWS CLI is appropriate for this particular deployment style, thanks to the eksctl command line utility, developed for AWS Kubernetes.

So, without further ado, let's proceed!

Part I. Architecture

1. CQRS

The CQRS principle is *a* natural extension of the microservices architecture. Although not an only way to work with microservices, it has its domain of applicability. Many an article has been written to articulate the microservices architecture, which we won't recapitulate. Instead, we only will highlight its most essential features by means of a metaphor. The gist of the entire architecture is: an application may use several microservices, each of which may be used by other applications, too—think of this as the warp and the weft of the software fabric, in a way. Loose coupling between microservices is necessary to prevent the applications from stepping on each other's toes.

This, in turn, demands a separation between the data storage managed by each microservice. This separation can be "physical" (different databases on different servers), or it can be logical (different tables in the same database, but each managed by a single microservice only). We may not even know whether the data managed by different microservices resides on the same server or not, as in the case of managed distributed databases such as AWS DynamoDB. Whichever is the case, the key is that each storage segment *is written to* by a single microservice only.

The natural next step then is to recognize that the same data doesn't have to be read in the same way as it is written. Instead, different "read views" can be presented by potentially different microservices. The advantage of this approach is that, while each data segment is only written to by a single microservice, the read microservices can assemble the read views from different write segments to present an overall, unifying picture. This is safe, because the data is read-only, so the different applications cannot step on each other's toes modifying it.

The separation between the write and read sides—or the command and query sides, respectively—underlies the CQRS principle. Commands are used to write data, whereas queries are used to retrieve the data from the read-only consolidated views, and the two are implemented differently in software—in different databases and possibly by different microservices entirely.

2. Event Sourcing

There are many ways to implement the CQRS principle, but Event Sourcing has emerged as a popular approach. The idea is to implement the command (write) side as a journal, recording the commands themselves—well, strictly speaking, the commands are transformed into events (an aggregate created, a field of an aggregate modified, an aggregate deleted, and so on), which are then recorded. Each aggregate's current state can then be reconstructed on demand from its history, and any of the prior states can be reconstructed too, if necessary. In practice, snapshots of each aggregate's state are periodically taken, and the current state is reconstructed from the latest snapshot and the events recorded since then—for the sake of performance. But the general idea is basically this: to store each aggregate in the form of its history.

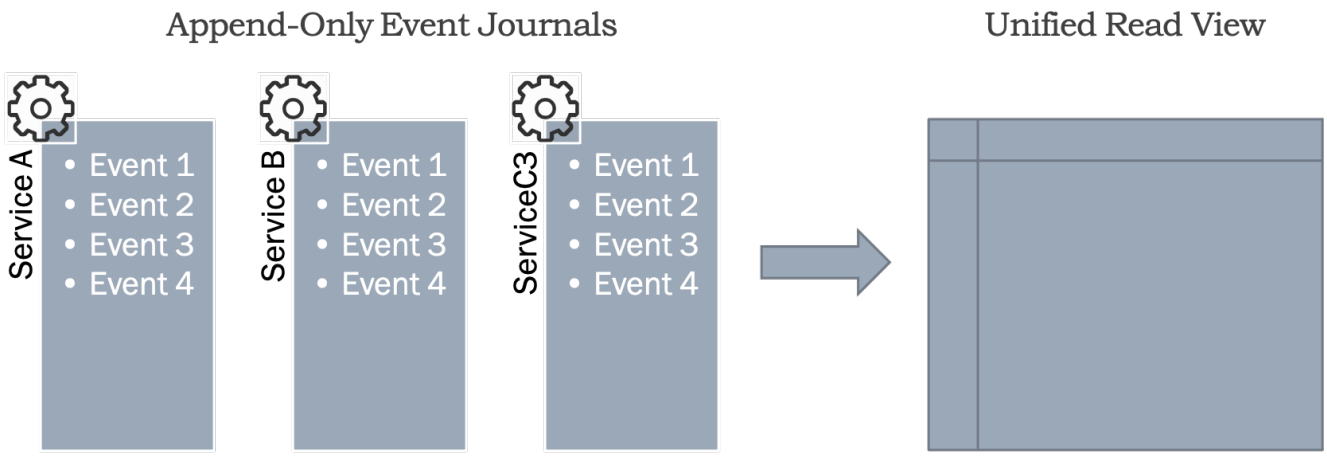


Figure 1. Event Sourcing as CQRS

The write-side storage, therefore, becomes append-only. This is very efficient, so it is well suited to write-heavy applications. Another advantage is that nothing is ever lost. In the CRUD approach, an application typically modifies an aggregate's state in place, replacing its prior state with a new one. With event sourcing, the prior state is never lost, as the entire history is preserved. Not only is this useful for auditing (another advantage of event sourcing), but the history can serve as input to data mining applications.

For example, if a microservice records user requests for different kinds of quotes (for health insurance or mortgages, and the like—what is called a *lead,* in the industry parlance), typically handling hundreds of thousands entries per second (a write-heavy application), then, besides the immediate need to save every lead and then analyze and trade them, another application may be interested in finding various patterns in the lead data. Thus, the journal itself—the command side of the lead-processing microservice—can serve as the query side of another, data mining microservice.

A seeming disadvantage is that it is hard to query the journal—for example, to search among aggregates. But that is natural, because the command side is not meant to serve as the query side. Instead, the query, read-only side is implemented differently, often in the form of tables in a relational database or similar—pretty much the way the storage would have looked in a CRUD scenario. As the commands arrive and the corresponding events are appended to the journal, the events are intercepted by event processors, which then update the read view.

As a result, there may be a delay between the events being recorded in the journal and them being reflected in the read view. This is often referred to as *eventual consistency*. However, this terminology is not, strictly speaking, accurate. The read view itself can, at any given moment, be strongly consistent, if updated via ACID transactions. It may merely not reflect a few more recent events, is all.

Another advantage of the separation between the journal and the read view is that they can be scaled differently. Moreover, with the journal serving as the source of truth, the entire read-side database can be wiped out and then reconstructed from the history of events as it is recorded in the journal.

Based on this discussion, the applications that would benefit from event sourcing are more likely to be write-heavy and/or embedded in a matrix of multiple microservices. The applications that must keep records of their history—for example, for the purposes of auditing and/or data mining—would also benefit. However, for the applications that are expected to immediately reflect all changes—say, in the UI—event sourcing may not be the best choice, because of the eventual consistency delay.

It so happens that the Tour of Heroes belongs to the latter kind of application. Nevertheless, for the purposes of this tutorial, we will imagine a stream of heroes being entered at the rate of hundreds of thousands per second, and we will agree to tolerate a brief delay between creation/update/deletion of a hero and that event being reflected in the UI's view.

Another potential complication can be the presence of uniqueness and other complex constraints in the domain model. In the Tour of Heroes, the heroes' names are not unique; only the IDs are, which are not supplied by the user. If, however, the names *were* unique, then the only way to check for any existing heroes with a given name when creating a new one is to query the read side. However, the read side may not reflect the latest changes. So, while the write side applies the command, the further update of the read side may fail.

In cases like this, it helps to have a domain model that includes the concept of a submitted but not yet applied command, as well as a special microservice that subscribes to messages from the read side on whether an event update succeeds or fails, then sending commands to the write side in order to approve or reject the previously submitted commands. It is important also to make sure that the event history replay is deterministic, which is non-trivial with multiple event processors updating the read side in parallel. But we won't go into that for our Tour of Heroes microservice.

So, let's proceed and see how event sourcing can be implemented in practice.

3. Akka Persistence

One approach to implementing Event Sourcing is to use Akka Persistence. Pioneered by Jonas Bonér, one of the co-authors of the Reactive Manifesto, Akka is an ambitious undertaking to develop a platform for building reactive applications. Akka is developed for Scala and Java. In its foundation is the concept of Actor.

In a nutshell, actors are lightweight objects that exchange messages among themselves and act on them. Although an actor is an instance of the **Actor** class, its method cannot be directly called, because an actor is never returned by any call. Only instances of **ActorRef** are returned. The only way to make an actor do anything is to send the actor a message. A special method in the actor takes the message and does different things depending on the class of the message object and its contents. Among other things the actor can do is to send a message back and/or to any other actor in the system (including remote actors, executing on a completely different server in the same Akka cluster).

Actors make asynchronous processing easy. Each actor's mailbox is single-threaded, so there is no contention. If a return message is expected, it is wrapped in a **Future**, which becomes complete once the actor sends back a return message. It is easy to reason about actors by imagining them as employees of a company. There, too, everything is done ultimately by communication—by exchanging messages. This way, it is possible to build easy-to-manage asynchronous software without any need for blocking calls and no "callback hell."

Akka actors can be organized in many different topologies. One popular one is that of a manager actor supervising a set of worker actors—and no wonder, given the company metaphor above. For an example of such actor topology, see a tutorial on developing a simple web crawler with Akka. But other topologies are possible. We are only in the budding stages of using actors, so more involved, more interesting topologies are yet to come.

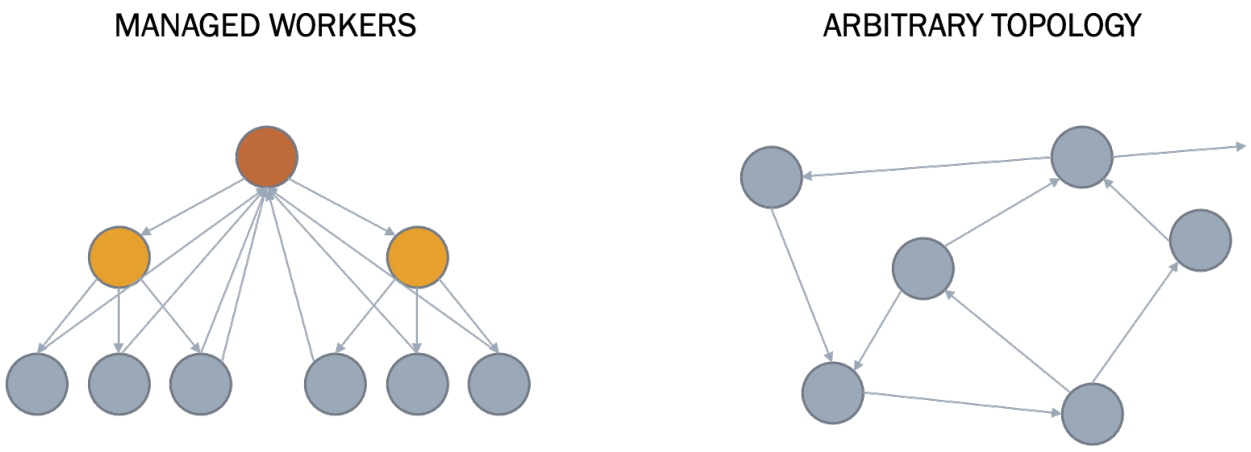


Figure 2. Akka Actor Topologies

Akka Persistence uses another popular actor topology, designed with event sourcing in mind. For every aggregate in the system (a hero, in the Tour of Heroes), there is a dedicated actor controlling access to it. Any commands to change the state of an aggregate are sent as messages to the corresponding actor. Since the actor's mailbox is single-threaded, such actors, in essence, protect shared mutable state from thread contention.

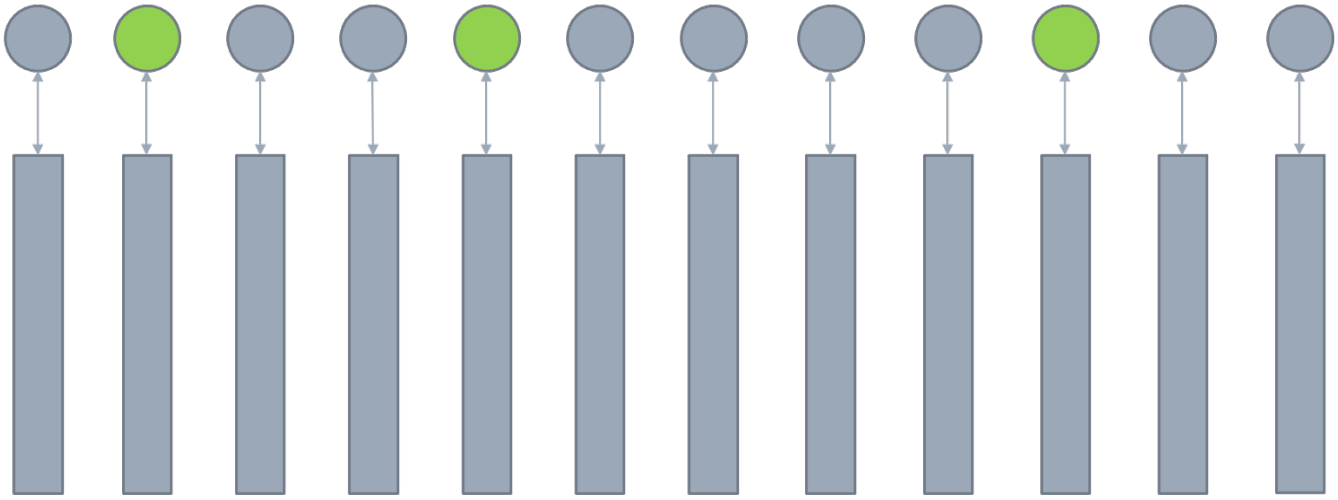


Figure 3. Akka Persistence

It may seem, at first glance, that too many actors like this could kill performance. But this is not true. Even though there may be millions or billions of aggregates in the system (remember, we assume a heavy load of hundreds of thousands of heroes being entered into the system per second?), only a limited number of actors per node in the cluster would be active at any given time. Actors are lightweight objects, given thread to execute on only when active. After some configurable period of inactivity, an actor is "passivated" (and in many load patterns, any given aggregate is only modified rarely; new aggregate creation is more typical, as in the example of leads above). The same can happen if the node runs out of threads in its pool—although, when autoscaling is properly configured, that should lead to the number of nodes scaling up, relieving the load on the affected node.

Akka Persistence is used in the Lagom framework, which is the one that we will use in this tutorial.

4. Lagom Framework

The Lagom framework has been developed by Lightbend, the company founded by the creator of the Scala programming language. For all of its advantages, Event Sourcing has been hard to implement (as is often the case with new technologies), so Lagom is designed to make Event Sourcing easy—so easy, in fact, that Lagom has been billed as something even a beginner can easily use.

Lagom applications can be developed in Scala and/or in Java (including a mix of the two), but we will use Scala to illustrate the development process, for conciseness and clarity. Scala is very similar to Java (they are sister languages, both running in JVM), but it follows functional programming more consistently, and it is easier to develop in. Comments on the Java version can be found toward the end of this whitepaper.

Lagom is built on top of another popular framework: Play, also developed by Lightbend for both Scala and Java. However, whereas Play is an MVC framework, Lagom is not. It is designed to implement microservices and uses Akka Persistence for Event Sourcing. This, however, won't prevent us from taking advantage of some useful Play modules—in particular, Play Json, which makes Json serialization and deserialization a breeze. We will also be able, like in Play, to have our application automatically pick up any changes to the code and configuration files when running in the development mode. But many other things, such as routing, are different in Lagom.

Any given database requires a journal and snapshot-store plugin to be used with Akka Persistence. Of several such plugins already developed, Lagom comes out of the box with two: one for Apache Cassandra and another for relational databases. (Unfortunately, a plugin for AWS DynamoDB is not yet ready for prime time.) Apache Cassandra is a great choice for Event Sourcing, especially for the write side, since it's a distributed database optimized for heavy write load. Since Lagom comes with an embedded Cassandra server for the development mode, we will start with using it for both write side and read side.

So, let's create a Lagom project and set up our development environment.

Part II. Development Mode

5. Development Environment

Besides having a JDK installed (Java 8 or later, but we will use Java 11, because Java 8 has reached the end of life), Lagom requires SBT (Simple Build Tool) for Scala, Java, and a mix thereof or Maven for Java-only code. While Maven is supported (for legacy projects, perhaps?), it is far less flexible; in particular, some of the easy build customization we'll do later in this project are hard to achieve with it. So, Lightbend's own templates use SBT for Scala and Java alike. We also will use SBT for both Scala and Java implementations of our microservice, although only the Scala project will be discussed in detail (for some important notes on the Java version, see the end of this whitepaper).

SBT is called "simple" not because all SBT projects are necessarily simple (one can go overboard with any tool) but because, when using SBT, you "simply" write your project configuration as code, pretty much like the rest of your code, which makes it flexible and very expressive—not in some clunky XML format.

Lagom recommends SBT 1.2.1 or later. We will use the latest version available at the moment: 1.3.8. SBT can be downloaded from here, while instructions for installing it can be found here. This is enough to set up the project. We don't even need to download Scala, as this will be done automatically by SBT, along with all other dependencies.

Choose a location for the project, under which a `toh-lagom` folder will be created (but don't create it yet). As the next step, we could have created an initial `build.sbt` file in that folder by hand, but it is easier—and instructive—to initialize your project by using a pre-defined Lagom template. Run the following command to create a Scala project:

```
sbt new lagom/lagom-scala.g8
```

For a Java project, replace `scala` with `java`. While we will not go into the same detail for the Java version of the Tour of Heroes microservice, the final code is available in our GitHub repository, and we will talk toward the end of this whitepaper about some important differences between the Java and Scala implementations. One of them is that here we take advantage of Scala's compile-time dependency injection, whereas a Java project relies on Guice for runtime injection.

The template will ask you for some parameters. You can accept the defaults of provide your own. We will use the following values:

- `name`: toh-lagom
- `organization`: com.chariotsolutions
- `version`: 0.0.1 (to start with)
- `package`: com.chariotsolutions.tohlagom
- `lagom-version`: 1.6.1 (currently, the latest)

This creates a copy of the "Hello, world!" project in the `toh-lagom` folder. We will modify it in a

moment, but it can be instructive to take a look at how it is organized. The following files and folders under `toh-lagom` are of interest:

- `toh-lagom-api`: This folder contains the API subproject.
- `toh-lagom-impl`: This folder contains the implementation subproject.
- `toh-lagom-stream-api`: This folder contains the API subproject for streaming microservices.
- `toh-lagom-stream-impl`: This folder contains the implementation subproject for streaming microservices.
- `project`: This is where code lives that is invoked from `build.sbt`.
- `build.sbt`: This file defines the project.

We would have used the streaming-style subprojects had we been working with a WebSockets client. Then, we would have been able to have endpoints that return streams. Unfortunately, the Tour of Heroes frontend uses an HTTP client, so we will have to return already materialized collections. Thus, the streaming subprojects `toh-lagom-stream-api` and `toh-lagom-stream-impl`, once examined for educational purposes, can be deleted and `build.sbt` modified to remove any references to them.

At the moment, the `project` folder only contains two files, since we don't have any custom code that must be executed from `build.sbt`:

- `build.properties`: defines the SBT version (upgrade it to 1.3.8).
- `plugins.sbt`: defines SBT plugins used by the project.

Currently, `plugins.sbt` defines only the Lagom plugin:

project/plugins.sbt

```
addSbtPlugin("com.lightbend.lagom" % "lagom-sbt-plugin" % "1.6.1")
```

Later, we will add a plugin to upload a generated Docker image to Amazon ECR (Elastic Container Registry), but for now we will leave `plugins.sbt` as is.

At this stage, besides removing the streaming-style subprojects, we will only make minor changes in `build.sbt`. Namely, we will redefine these library dependencies:

build.sbt

```
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3" % "provided"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1" % Test
```

as follows:

build.sbt

```
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
```

```
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"
```

The dependencies follow the usual Maven conventions but expressed with SBt syntax. Here, for example, `com.softwaremill.macwire` is the group ID, `macros` is the artifact ID, `2.3.3` is the version, and `provided` is the scope. The double-percent sign injects the current Scala version into the group ID, for Scala libraries—unnecessary for Java libraries.

Then, we will use `macwire % Provided` and `scalaTest % Test` where previously `macwire` and `scalaTest` were used, respectively. First, it is cleaner to use a constant `Provided` instead of a literal string. Second, the same library may potentially be used with different scopes in different places. When we add integration tests later, we will need to differentiate between `scalaTest % Test` and `scalaTest % IntegrationTest`. In general, it is a good practice to define often-reused dependencies without scopes, adding the scopes as necessary later on.

After this is done, the first iteration of `build.sbt` should look like this:

build.sbt

```
name in ThisBuild := "toh-lagom"
organization in ThisBuild := "com.chariotsolutions"
scalaVersion in ThisBuild := "2.13.1"
version in ThisBuild := "0.0.1"

val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"

lazy val `toh-lagom` = (project in file("."))
  .aggregate(`toh-lagom-api`, `toh-lagom-impl`)

lazy val `toh-lagom-api` = (project in file("toh-lagom-api"))
  .settings(libraryDependencies ++= Seq(lagomScaladslApi))

lazy val `toh-lagom-impl` = (project in file("toh-lagom-impl"))
  .enablePlugins(LagomScala)
  .dependsOn(`toh-lagom-api`)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      macwire % Provided
    )
  )
  .settings(
    lagomForkedTestSettings,
    libraryDependencies ++= Seq(
      lagomScaladslTestKit,
      scalaTest % Test
    )
  )
```

Admittedly, all arguments to the `settings` calls could be combined into a single call. But we have

separated the test settings from the application code settings, for clarity.

The back-tick quotes for the project values are used in order to allow hyphens in the identifiers, which otherwise must follow the Java rules.

As you can see, the `toh-lagom` project is defined as an aggregate of two subprojects. Of those, `toh-lagom-api` depends only on Lagom's only Scala API, whereas `toh-lagom-impl` depends on `toh-lagom-api`, on Cassandra components, on the `macwire` library that provides Scala macros for compile-time dependency injection, and on `scalaTest` for testing purposes. We have removed a dependency on Kafka, since we won't use it in our project.

6. The API Subproject

The `toh-lagom-api` subproject, as generated from the template, has a single package, `com.chariotsolutions.tohlagom.api`, with a single file in it, `TohlagomService.scala`. The file includes a trait (corresponding to a Java interface) with declarations of endpoints and route bindings that tell which endpoint is invoked with which parameters for which HTTP request. The file also defines the classes into/from which the HTTP requests/responses are to be (de)serialized.

However, since it currently contains code for a "Hello, world!" project (with the capability to change the greeting message), we will need to make quite a few changes.

Before we do, however, take a note of the sample code that shows how one can use Apache Kafka producers in a Lagom microservice. Besides registering event providers to update the read side, Lagom makes it easy to keep other microservices abreast of any events by publishing them to a configured Kafka topic. Lagom microservices can also use Kafka consumers to glean events from Kafka topics. This can be done in addition to or instead of using event processors, constructing the read side from Kafka topics.

Lagom has been developed also with an eye to potentially hook up AWS Kinesis (the closest AWS counterpart to Kafka) and Amazon SQS (for simpler scenarios) for the same purposes, although only experimental work has been done for integrating Lagom with those: see here for Lagom Kinesis provider and here for Lagom SQS provider.

You can find out more about how to work with Kafka in Lagom here (for Scala) and here (for Java). But for the purposes of our tutorial, we will not use Kafka, so we will remove the corresponding sample code.

Let's rename the automatically generated trait `TohlagomService` to `TourOfHeroesService` (and the file, as well). Since its companion object defines only the name of the Kafka topic, we won't need it, so it can be removed.

In the Tour of Heroes, each hero has an ID and a name. The ID is an integer; the in-memory stubs provided with the Angular's tutorial has the IDs starting at 11 and rising sequentially for every new hero created. However, in Lagom, every aggregate must have a string ID, as random as possible. This is because event-sourced microservices are often designed to handle a heavy write load and because distributed databases under heavy load don't handle global sequences well.

Usually, a UUID is used for aggregate IDs in Lagom. However, since the Tour of Heroes frontend demands integer IDs (using anything else results in errors), we will use random integers, converted into strings that are padded with zeroes on the left up to the ten-digit length—this is because the frontend doesn't adjust the length of the ID display depending on the number of digits in the integer, and we want to avoid any changes in the frontend besides turning off the in-memory stubbing. Now, random integers are not as random as UUIDs, so don't do this for a serious production microservice! But for our purposes, this should suffice.

Let's now create the `Hero` class, whose instances will be serialized in HTTP responses, and the `NewHero` class, whose instances will be passed with HTTP requests for creation of new heroes.

toh-lagom-api/src/main/scala/com/chariotsolutions/tohlagom/api/TourOfHeroService.scala

```
case class Hero(id: String, name: String)
case class NewHero(name: String)
```

In Java, each class would have required a separate file with much boilerplate: a public constructor and the getters for each field, as well as mamny annotations for Json (de)serialization. But in Scala, each is just one line.

These classes also need to be (de)serialized to/from Json. Lagom uses Play Json for this, so we just create a companion object for each class with an implicit instance of Json format in it:

toh-lagom-api/src/main/scala/com/chariotsolutions/tohlagom/api/TourOfHeroService.scala

```
import play.api.libs.json.{Format, Json}

object Hero {
  implicit val format: Format[Hero] = Json.format
}

object NewHero {
  implicit val format: Format[NewHero] = Json.format
}
```

That's all there is to it, besides registering each serializer in the `toh-lagom-impl` subproject, which we will do later. The `Json.format` Scala macro automatically detects all fields in the class to be serialized and does its job right, unless some custom handling must be implemented (which we don't require, in this case).

Now we will define the Tour of Heroes endpoints. Here's how the method to create a new hero is defined:

toh-lagom-api/src/main/scala/com/chariotsolutions/tohlagom/api/TourOfHeroService.scala

```
  def createHero(): ServiceCall[NewHero, Hero]
```

The first type argument in `ServiceCall` is the type of the POST payload, and the second type argument is that of the response. Overall, we need to implement the following HTTP endpoints:

- `GET /api/heroes`: list all heroes.
- `PUT /api/heroes`: rename an existing hero.
- `POST /api/heroes`: create a new hero.
- `GET /api/heroes/[hero-id]`: fetch a hero by ID.
- `GET /api/heroes?name=[name-prefix]`: search for heroes whose name starts with a given prefix.
- `DELETE /api/heroes/[hero-id]`: delete an existing hero.

Once we declare all necessary methods, we bind them to the HTTP endpoints in the `TourOfHeroesService.description` method. The complete `TourOfHeroesService.scala` file, with all comments stripped for concise display on this page, looks as follows:

toh-lagom-api/src/main/scala/com/chariotsolutions/tohlagom/api/TourOfHeroService.scala

```
package com.chariotsolutions.tohlagom.api

import akka.{Done, NotUsed}
import play.api.libs.json.{Format, Json}
import com.lightbend.lagom.scaladsl.api.transport.Method
import com.lightbend.lagom.scaladsl.api.{Descriptor, Service, ServiceCall}

case class Hero(id: String, name: String)
case class NewHero(name: String)

object Hero {
  implicit val format: Format[Hero] = Json.format
}

object NewHero {
  implicit val format: Format[NewHero] = Json.format
}

trait TourOfHeroesService extends Service {
  def heroes(): ServiceCall[NotUsed, Seq[Hero]]
  def search(name: String): ServiceCall[NotUsed, Seq[Hero]]
  def fetchHero(id: String): ServiceCall[NotUsed, Hero]
  def createHero(): ServiceCall[NewHero, Hero]
  def changeHero(): ServiceCall[Hero, Done]
  def deleteHero(heroId: String): ServiceCall[NotUsed, Done]

  final def descriptor: Descriptor = {
    import Service._
    named("toh-lagom").withCalls(
      restCall(Method.GET, "/api/heroes", heroes _),
      restCall(Method.PUT, "/api/heroes", changeHero _),
      restCall(Method.POST, "/api/heroes", createHero _),
      restCall(Method.GET, "/api/heroes/?name", search _),
      restCall(Method.GET, "/api/heroes/:heroId", fetchHero _),
      restCall(Method.DELETE, "/api/heroes/:heroId", deleteHero _)
    ).withAutoAcl(true)
  }
}
```

The `descriptor` method is declared in the `Service` base trait, so it must be implemented here. In Java, the convention is to use the `@Override` annotation in such cases, whereas Scala has the `override` method modifier. However, `override` is only required by the compiler if a superclass actually has the method defined, so we really override the thing. If it's only declared, it is not required, and I follow the

> school of thought that uses the `override` modifier only when it is required. So I have removed it from the template-generated file.

Now we turn to implementing the methods we have just declared and bound to HTTP endpoints.

7. Hero Commands

Recall that, in an event-sourced microservice, any change to an aggregate is accomplished by means of sending a command. As generated from the template, the commands for the sample project we started with can be found in the `TohlagomAggregate.scala` file, in the `com.chariotsolutions.tohlagom.impl` package of the `toh-lagom-impl` subproject. However, those are for the "Hello, world!" purposes, so we must replace them.

For clarity, we will move all commands into a separate file, `HeroCommand.scala`. We will define the `HeroCommand` trait that all command must implement as a sealed trait, meaning that it can only be extended in this file. Each command must include the necessary data for the change, as well as a reference to an actor that accepts messages of the `Confirmation` type. This is the actor that awaits a response from the command processor.

A hero can be created, renamed, and deleted. In addition, we will provide a command to fetch a hero, although *we will only use it for unit testing purposes*. Technically speaking, it is possible to fetch the current state of a hero by sending such a command to the actor responsible for that particular hero, since this query doesn't involve other heroes. However, in production, we will use a read-side query to fetch even a single hero, for the following reasons:

- Fetching a hero is a read-only operation, so it belongs with the read side, conceptually.
- The read side may not reflect a few latest changes, but its state is internally consistent. We don't want to be able to fetch a hero that is not yet in the list of all existing heroes.
- A fetch command may have to bring an uninitialized actor into the active state, which would mean restoring its state from its latest snapshot and subsequent history—an unnecessary expense when the hero's state is available from the read side.

The `FetchHero` command must be responded to with `HeroState` instead of `Confirmation`, which will be defined later.

The `HeroCommand.scala` file looks as follows (again, stripping the comments for conciseness on this page):

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroCommand.scala

```
package com.chariotsolutions.tohlagom.impl

import akka.actor.typed.ActorRef

sealed trait HeroCommand
case class CreateHero(name: String, replyTo: ActorRef[Confirmation]) extends
HeroCommand
case class ChangeHero(newName: String, replyTo: ActorRef[Confirmation]) extends
HeroCommand
case class DeleteHero(replyTo: ActorRef[Confirmation]) extends HeroCommand
case class FetchHero(replyTo: ActorRef[HeroState]) extends HeroCommand // for unit
testing only
```

This is all there is to do for this file. Again, each line would have been a separate file, or a nested class, with much boilerplate code, in Java. Commands must be serialized differently from other objects, because they have fields of type `ActorRef`. Instead of using Play Json, we will use Akka Jackson for command serialization, which is accomplished by means of the following setting in `application.conf` (in the `toh-lagom-impl/src/main/resources` folder):

toh-lagom-impl/src/main/resources/application.conf

```
akka.actor.serialization-bindings {
  "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
}
```

We will define the `Confirmation` trait in the `Confirmation.scala` file, in the same package.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/Confirmation.scala

```
package com.chariotsolutions.tohlagom.impl

import play.api.libs.json.{Format, Json}

sealed trait Confirmation

object Confirmation {
  implicit val format: Format[Confirmation] = Json.format
}

case object Accepted extends Confirmation {
  implicit val format: Format[Accepted.type] = Json.format
}

case class Rejected(reason: String) extends Confirmation

object Rejected {
  implicit val format: Format[Rejected] = Json.format
}
```

Here, we use Play Json serialization.

`Accepted` is a singleton object, not a class, so it is its own object companion. The code for `Confirmation` is much simpler than what was generated from the template in `TohlagomAggregate.scala`, which can be removed.

8. Hero Events

When commands are processed, events must be persisted in the journal. It is possible for multiple events to be the result of a single complex command, but the Tour of Heroes is a simple application. Our events will correspond to the commands pretty much one-to-one. Nevertheless, there is an additional matter to be addressed—namely, that of event tagging.

When we begin defining the `HeroEvent` trait in `HeroEvent.scala`, we must implement the `aggregateTag` method of the base trait:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEvent.scala

```
import play.api.libs.json.{Format, Json}
import com.lightbend.lagom.scaladsl.persistence.{AggregateEvent, AggregateEventTag}

sealed trait HeroEvent extends AggregateEvent[HeroEvent] {
  def aggregateTag = HeroEvent.Tag
}
```

Here, `HeroEvent.Tag` is defined in the companion object (Java developers can think of the companion object as the static scope of the class), as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEvent.scala

```
object HeroEvent {
  implicit val format: Format[HeroEvent] = Json.format
  val NumShards = 2
  val Tag =
    if (NumShards > 1) AggregateEventTag.sharded[HeroEvent](NumShards)
    else AggregateEventTag[HeroEvent]
}
```

Lagom documentation explains event sharding as follows. Having a single shard for all events is the simplest way to persist them, but you can only consume the events one at a time, which may be a bottleneck to scale. If you expect events to only occur in the order of a few times per second, then this might be fine, but if you expect hundreds or thousands of events per second, then you may want to shard your read-side event processing load.

Sharding can be done in two ways, either manually by returning different tags based on information in the event, or automatically by returning an `AggregateEventShards` tag, which will tell Lagom to shard the tag used based on the entity's persistence ID. It's important to ensure all the events for the same entity end up with the same tag (and hence in the same shard); otherwise, event processing for that entity may be out of order, since the read-side nodes will consume the event streams for their tags at different paces.

When you shard events, you need to decide upfront how many shards you want to use. The more shards, the more you can scale your service horizontally across many nodes. However, shards come at a cost, as each additional shard increases the number of read-side processors that query your

database for new events. It is very difficult to change the number of shards without compromising the ordering of events within an entity, so it's best to work out upfront what the peak rate of events you expect to need to handle over the lifetime of the system will be, then work out how many nodes you'll need to handle that load, and then use that as the number of shards. For illustration purposes, let's choose a small number that's not one—to wit, two. This means that we will have two event processors working in parallel.

Note how the Play Json serialization format for `HeroEvent` is defined in the companion object. This is how it is done for Play Json in Scala, in general. The rest of `HeroEvent.scala` is fairly straightforward.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEvent.scala

```
case class HeroCreated(name: String) extends HeroEvent
case class HeroChanged(newName: String, oldName: String) extends HeroEvent

case object HeroDeleted extends HeroEvent {
  implicit val format: Format[HeroDeleted.type] = Json.format
}

object HeroCreated {
  implicit val format: Format[HeroCreated] = Json.format
}

object HeroChanged {
  implicit val format: Format[HeroChanged] = Json.format
}
```

We do include `oldName` in the `HeroChanged` event. Although it is not required to reconstruct the current state of a hero from the event history, this information is good to have for auditing purposes. Normally, events would also have the user performing the action, but the frontend doesn't have the concept of a user, so we don't either. Timestamps are always included in each event, as part of the base trait.

9. State and Behavior

As discussed before, Akka Persistence uses actors to manage shared mutual state of the aggregates—in our case, the hero aggregates. Now it's time to define the actual state that will be thus managed. Unlike the template-generated code, we will put the `HeroState` class into a file of its own, `HeroState.scala`, in the same `com.chariotsolutions.tohlagom.impl` package.

Besides the hero's name, the `HeroState` class will also have the boolean field `valid`, which will be `true` for existing heroes and `false` for deleted heroes and for the heroes in the initial state (see below). Since the write side holds the entire history of events, it contains the information about all heroes that were ever created, even the deleted ones—great for auditing.

`HeroState` doesn't include the hero's ID, because it is assigned not to the state but to the aggregate.

The `HeroState` class will also have two methods: a command handler and an event handler. These will become important when `HeroState` is declared as the `State` type parameter of Lagom's `EventSourcedBehavior`. The command handler persists events based on the received command, after performing some simple validity checks. The event handler updates the aggregate's state based on the events persisted. The implementation below is self-explanatory.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroState.scala

```
package com.chariotsolutions.tohlagom.impl

import play.api.libs.json.{Format, Json}
import akka.cluster.sharding.typed.scaladsl.EntityTypeKey
import akka.persistence.typed.scaladsl.{Effect, ReplyEffect}

case class HeroState(name: String, valid: Boolean) {
  def applyCommand(cmd: HeroCommand): ReplyEffect[HeroEvent, HeroState] = cmd match {
    case CreateHero(name, replyTo) =>
      if (valid) Effect.reply(replyTo)(Rejected("The hero already exists."))
      else Effect.persist(HeroCreated(name.toLowerCase)).thenReply(replyTo) { _ =>
Accepted }
    case ChangeHero(newName, replyTo) =>
      if (!valid) Effect.reply(replyTo)(Rejected("The hero is in invalid state."))
      else Effect.persist(HeroChanged(newName.toLowerCase, name)).thenReply(replyTo) {
_ => Accepted }
    case DeleteHero(replyTo) =>
      if (!valid) Effect.reply(replyTo)(Rejected("The hero has already been
deleted."))
      else Effect.persist(HeroDeleted).thenReply(replyTo) { _ => Accepted }
    case FetchHero(replyTo) if valid => // only used for unit testing
      Effect.reply(replyTo)(this)
    case _ =>
      Effect.noReply
  }
```

```
  def applyEvent(event: HeroEvent): HeroState = event match {
    case HeroCreated(newName) => HeroState(newName, valid = true)
    case HeroChanged(newName, _) => HeroState(newName, valid)
    case HeroDeleted => HeroState(name, valid = false)
    case _ => this
  }
}

object HeroState {
  def initial = HeroState("", valid = false)
  val typeKey = EntityTypeKey[HeroCommand]("HeroAggregate")
  implicit val format: Format[HeroState] = Json.format
}
```

In the same file, we will also put the `HeroState` companion object (which Java developers can think of as the class's static scope). You must have recognized the Json serialization format. The `initial` method is used to initialized the aggregate if no snapshot state is found. The events from the aggregate's history are then applied to this initial state. But `typeKey` requires some explanation.

The actors managing the aggregates are sharded inside the Akka cluster. When sharding actors and distributing them across the cluster, each aggregate is namespaced under a `typeKey` that specifies a name and also the type of the commands that a sharded actor can receive—in our case, "HeroAggregate" and `HeroCommand`, respectively. This becomes important when working with aggregates of many types.

Each aggregate in Lagom is represented by an instance of `EventSourcedBehavior[Command, Event, State]`, which extends `Behavior[Command]`. Behaviors define how actors... well, *behave*. We will put the methods to create behaviors into the `HeroBehavior` object. Java developers can think of them as static methods.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroBehavior.scala

```
object HeroBehavior {
  def create(entityContext: EntityContext[HeroCommand]): Behavior[HeroCommand] = {
    val persistenceId = PersistenceId(entityContext.entityTypeKey.name, entityContext
.entityId)

    create(persistenceId).withTagger(
      AkkaTaggerAdapter.fromLagom(entityContext, HeroEvent.Tag)
    )
  }

  private[impl] def create(persistenceId: PersistenceId): EventSourcedBehavior
[HeroCommand, HeroEvent, HeroState] =
    EventSourcedBehavior.withEnforcedReplies[HeroCommand, HeroEvent, HeroState](
      persistenceId = persistenceId,
      emptyState = HeroState.initial,
      commandHandler = (state, cmd) => state.applyCommand(cmd),
      eventHandler = (state, event) => state.applyEvent(event)
    )
```

```
}
```

As you can see, we pass to `create` the command and event handlers that we have defined above in `HeroState`, as well as the `initial` method for creating the empty state. If you wonder where the hero's ID is here, it is passed as `entityContext.entityId`. The private method is created separately for use in unit tests, since it can be called without having an Akka cluster. This is also why it is declared as private to the `com.chariotsolutions.tohlagom.impl` package, to which the unit tests also will belong. The tagger `AkkaTaggerAdapter.fromLagom(entityContext, HeroEvent.Tag)` is defined in the Lagom-compatible way, so that the event processors and Kafka producers could locate and follow the event streams.

Now that we have implemented the write side, let's proceed with defining event processors for the read side.

10. Event Processors

Read-side event processors read the events persisted in the journal and update the read side accordingly. Each persisted event has an offset, and the event processors must keep track of the last offset processed, so as not to process the same event more than once. The good news is that if you use Cassandra or relational database read-side plugins that come with Lagom out of the box, then offset tracking is taken care for you. There are ways to define custom read-side event processors, however, which can be found in Lagom documentation.

According to our plan, we will use the Lagom's embedded Cassandra server for both write side and read side. Embedded Cassandra automatically starts when running in the development mode. So, here we will write a Cassandra event processor.

Let's start by defining a few constants in the companion object, in the `HeroEventProcessor.scala`file. Here, we define the name of the Cassandra table, the names of the two columns in that table (so as not to use literal strings below), and the event processor ID.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
object HeroEventProcessor {
  val Table = "hero"
  val IdColumn = "id"
  val NameColumn = "name"
  val EventProcessorId = "hero-offset"
}
```

The <codeHeroEventProcessor class will extend Lagom's `ReadSideProcessor[HeroEvent]` and have instances of `CassandraReadSide`, `CassandraSession`, and an implicit `ExecutionContext` (for composing futures) injected into it.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
class HeroEventProcessor(readSide: CassandraReadSide, session: CassandraSession)
                        (implicit ec: ExecutionContext) extends ReadSideProcessor
[HeroEvent] {
  import HeroEventProcessor._

  ???
}
```

`???` is valid Scala syntax. During runtime, it throws a `NotImplementedException` but the compiler will not complain. Useful as a placeholder while developing; for example, for stubbing a method: `def doSomething(arg: String): SomeType = ???`. Here we use it to indicate sections of code to be filled in.

It must implement the base class's `aggregateTags` method, which describes which tags this event processor is responsible for. If mnore than one tag is returned by this method, Lagom will automatically shard them across the Akka cluster. So, we will return all tags.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
def aggregateTags: Set[AggregateEventTag[HeroEvent]] = HeroEvent.Tag match {
  case tagger: AggregateEventTag[HeroEvent] =>
    Set(tagger)
  case shardedTagger: AggregateEventShards[HeroEvent] =>
    shardedTagger.allTags
}
```

Recall that, when using one shard, `HeroEvent.Tag` is an instance of `AggregateEventTag[HeroEvent]`, whereas if more shards are used, it is an instance of `AggregateEventShards[HeroEvent]`. Next, we must implement the following method, and we begin by instantiating a builder with our event processor ID:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
def buildHandler(): ReadSideProcessor.ReadSideHandler[HeroEvent] = {
  val builder = readSide.builder[HeroEvent](EventProcessorId)

  ???

  builder.build
}
```

Before calling `builder.build`, however, we must set various callbacks on the builder. The *global prepare* callback runs at least once across the cluster. Whereas all other internal Cassandra tables are automatically created by Lagom (although there is an option to turn the automatic creation of tables off, for manual setup), we must create the tables for the read view ourselves. In our case, we need just one `hero` table.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
  builder.setGlobalPrepare(() =>
    for {
      _ <- session.executeCreateTable(
        s"""CREATE TABLE IF NOT EXISTS $Table (
           |  $IdColumn TEXT,
           |  $NameColumn TEXT,
           |  PRIMARY KEY ($IdColumn)
           |)""".stripMargin)
      _ <- session.executeWrite(
        s"""CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix
           |  ON $Table ($NameColumn)
           |  USING 'org.apache.cassandra.index.sasi.SASIIndex'""".stripMargin)
    } yield Done
  )
```

Cassandra's CQL is similar to SQL in many ways, as you can see. However, there are differences. Thus, when we later implement our `search` endpoint by executing a `SELECT` statement, Cassandra

would have refused to execute a `LIKE` clause unless the custom index is created as above. This is the result of conscious design decisions, due to Cassandra's nature as a highly performant distributed database.

Here, we use Scala's multiline strings, for clarity (the triple-quote syntax). The underscore placeholders in the for-comprehension indicate that we are not interested in the value returned when the future completes.

The *prepare* callback is executed once per shard. Since the DataStax Cassandra driver's `PreparedStatement` is thread-safe (which cannot, unfortunately, be said of JDBC's `PreparedStatement`), we can prepare insert/update/delete statements in this callback, as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
  private val promiseInsert = Promise[PreparedStatement]
  private val promiseUpdate = Promise[PreparedStatement]
  private val promiseDelete = Promise[PreparedStatement]

  private def stmtInsert: Future[PreparedStatement] = promiseInsert.future
  private def stmtUpdate: Future[PreparedStatement] = promiseUpdate.future
  private def stmtDelete: Future[PreparedStatement] = promiseDelete.future

  def buildHandler(): ReadSideProcessor.ReadSideHandler[HeroEvent] = {
    ???

    builder.setPrepare { _ =>
      val insertStmt = session.prepare(s"INSERT INTO $Table ($IdColumn, $NameColumn)
VALUES (?, ?)")
      val updateStmt = session.prepare(s"UPDATE $Table SET $NameColumn = ? WHERE
$IdColumn = ?")
      val deleteStmt = session.prepare(s"DELETE FROM $Table WHERE $IdColumn = ?")

      promiseInsert.completeWith(insertStmt)
      promiseUpdate.completeWith(updateStmt)
      promiseDelete.completeWith(deleteStmt)

      for {
        _ <- insertStmt
        _ <- updateStmt
        _ <- deleteStmt
      } yield Done
    }

    ???
  }
```

Now we can add event handlers, as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
    builder.setEventHandler[HeroCreated] { element =>
      stmtInsert.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.name)
        List(stmt)
      }
    }

    builder.setEventHandler[HeroChanged] { element =>
      stmtUpdate.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.newName)
        List(stmt)
      }
    }

    builder.setEventHandler[HeroDeleted.type] { element =>
      stmtDelete.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        List(stmt)
      }
    }
```

The callbacks return lists of statements to the framework, but the statements are not executed in the callbacks themselves. This is because Lagom adds statements of its own to the list—specifically, updates to the internal table for managing offsets—and then executes them together as a logged batch (Cassandra's counterpart of transactions). This guarantees exactly-once processing for every event. If your events are idempotent, so that you can tolerate at-least-once processing, then you can execute the statements directly in the callbacks and return an empty list instead—but this is not recommended.

In its entirely, the first iteration of the `HeroEventProcessor.scala` file looks as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroEventProcessor.scala

```
package com.chariotsolutions.tohlagom.impl

import akka.Done
import scala.concurrent.{ExecutionContext, Future, Promise}
import com.lightbend.lagom.scaladsl.persistence.{AggregateEventShards,
AggregateEventTag, ReadSideProcessor}
import com.lightbend.lagom.scaladsl.persistence.cassandra.{CassandraReadSide,
CassandraSession}
import com.datastax.driver.core.PreparedStatement

object HeroEventProcessor {
  val Table = "hero"
  val IdColumn = "id"
```

```
  val NameColumn = "name"
  val EventProcessorId = "hero-offset"
}

class HeroEventProcessor(readSide: CassandraReadSide, session: CassandraSession)
                        (implicit ec: ExecutionContext) extends ReadSideProcessor
[HeroEvent] {
  import HeroEventProcessor._

  def aggregateTags: Set[AggregateEventTag[HeroEvent]] = HeroEvent.Tag match {
    case tagger: AggregateEventTag[HeroEvent] =>
      Set(tagger)
    case shardedTagger: AggregateEventShards[HeroEvent] =>
      shardedTagger.allTags
  }

  private val promiseInsert = Promise[PreparedStatement]
  private val promiseUpdate = Promise[PreparedStatement]
  private val promiseDelete = Promise[PreparedStatement]

  private def stmtInsert: Future[PreparedStatement] = promiseInsert.future
  private def stmtUpdate: Future[PreparedStatement] = promiseUpdate.future
  private def stmtDelete: Future[PreparedStatement] = promiseDelete.future

  def buildHandler(): ReadSideProcessor.ReadSideHandler[HeroEvent] = {
    val builder = readSide.builder[HeroEvent](EventProcessorId)

    builder.setGlobalPrepare(() =>
      for {
        _ <- session.executeCreateTable(
          s"""CREATE TABLE IF NOT EXISTS $Table (
             |  $IdColumn TEXT,
             |  $NameColumn TEXT,
             |  PRIMARY KEY ($IdColumn)
             |)""".stripMargin)
        _ <- session.executeWrite( s"""CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix |
ON $Table ($NameColumn) | USING 'org.apache.cassandra.index.sasi.SASIIndex'"""
.stripMargin) } yield Done ) builder.setPrepare { _ =>
      val insertStmt = session.prepare(s"INSERT INTO $Table ($IdColumn, $NameColumn)
VALUES (?, ?)")
      val updateStmt = session.prepare(s"UPDATE $Table SET $NameColumn = ? WHERE
$IdColumn = ?")
      val deleteStmt = session.prepare(s"DELETE FROM $Table WHERE $IdColumn = ?")

      promiseInsert.completeWith(insertStmt)
      promiseUpdate.completeWith(updateStmt)
      promiseDelete.completeWith(deleteStmt)

      for {
        _ <- insertStmt
        _ <- updateStmt
```

```
        _ <- deleteStmt } yield Done } builder.setEventHandler[HeroCreated] { element
=>
      stmtInsert.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.name)
        List(stmt)
      }
    }

    builder.setEventHandler[HeroChanged] { element =>
      stmtUpdate.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.newName)
        List(stmt)
      }
    }

    builder.setEventHandler[HeroDeleted.type] { element =>
      stmtDelete.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        List(stmt)
      }
    }

    builder.build
  }
}
```

Once an instance of `HeroEventProcessor` is registered with the service, all events persisted in the journal would translate into insert/update/delete statements on the Cassandra read side. Onward to implementing the service endpoints and registering the event processor!

11. Useful Helpers

First, we will need some helpers. Scala allows us to put them in the package scope. In the `com.chariotsolutions.tohlagom.impl` package, create a file called `package.scala` and define the *package object* as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/package.scala

```
package com.chariotsolutions.tohlagom

import akka.util.Timeout
import scala.util.Random
import scala.concurrent.duration._

package object impl {
  implicit val timeout = Timeout(10 seconds)
  def id2str(id: Int): String = f"$id%010d"
  def newId: String = id2str(Random.nextInt(Int.MaxValue))

  implicit class StringExt(str: String) {
    def toTitleCase = if (null == str) null else
      str.split(" ").map(_.toLowerCase.capitalize).mkString(" ")
  }
}
```

First, whenever we send a message to an actor and expect to receive a response (see the `ask` calls below), we need to pass an implicit `Timeout` argument. By defining it in the package scope, it becomes immediately available throughout the entire package.

Second, recall that we have decided to use random integers as hero IDs (again, don't do this for any serious project!), padding them with zeroes on the left to make them all the same ten-digits length for display purposes. The `id2str` function accomplishes that. See here for more information on Scala string interpolation.

Third, the `newId` function allows us to generate new hero IDs, both for the service implementation and for unit testing.

Last, we "extend" the Java `String` class with a `toTitleCase` method, which converts the string to lowercase and then capitalizes the first character of every word. This is done by means of an implicit class mechanism. When the Scala compiler encounters an expression `someString.toTitleCase` and determines that the `String` class doesn't have a real `toTitleCase` method, it looks for any implicit conversion of `String` to a class that does have that method. In our case, it finds the `StringExt` implicit class that takes a `String` argument in its constructor. So it knows to transform `someString.toTitleCase` into `new StringExt(someString).toTitleCase`.

The reason we need to transform strings to their title case is simple. As mentioned earlier, Apache Cassandra sacrifices some functionality for performance. One of these sacrifices is that there is no concept of case-sensitivity in Cassandra, as everything is stored as bytes and no case conversion functions are provided (again, for performance reasons). However, when we search heroes with a

prefix, we expect to find “Spiderman” in the dropdown when typing a lowercase “s.”

Thus, we will store all hero names in lowercase in Cassandra and transform all prefixes to lowercase when searching. But we will display the names in their title case.

Finally, let’s suppress compiler warnings about implicit conversions and postfix notation: such as the one in `10 seconds`, which is equivalent to `10.seconds`, where the `seconds` method has been implicitly added. One way to do this is to add `import scala.language.implicitConversions` and `import scala.language.postfixOps` in the files where these optional language features are used. But if you expect to use them often, a better way is to allow them throughout the entire project by adding the following Scala compiler options to `build.sbt`:

build.sbt

```
scalacOptions in ThisBuild ++= Seq(
  "-language:implicitConversions",
  "-language:postfixOps"
)
```

This is what we will do.

12. Implementing the Service

We will now implement the `TourOfHeroesService` trait from the API subproject. We begin by defining `TourOfHeroesServiceImpl` in the namesake file in `com.chariotsolutions.tohlagom.impl` as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
class TourOfHeroesServiceImpl(sharding: ClusterSharding)
                              (readSide: ReadSide, cassandraReadSide:
CassandraReadSide, session: CassandraSession)
                              (implicit ec: ExecutionContext) extends
TourOfHeroesService {
  import HeroEventProcessor._
  readSide.register[HeroEvent](new HeroEventProcessor(cassandraReadSide, session))

  ???
}
```

The constructor arguments are injected by the application loader (as we will see soon). Let us create two groups of non-implicit arguments—the ones related to the write side, followed by the ones related to the read side—but this is optional. A single group of arguments would work just as well, with arguments listed in any order whatsoever.

In the body of the class, we import such constants from `HeroEventProcessor` as the name of the read-side table and the names of its columns, so as not to use literal strings below. We also instantiate a `HeroEventProcessor` from the Cassandra session and register it with the injected read side.

Next, we implement the endpoints that call the read side. Recall that we have decided to do that even for fetching a single hero.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
  private def performRead(query: String): Future[Seq[Hero]] =
    session.selectAll(query).map(_.map(row => Hero(
      row.getString(IdColumn),
      row.getString(NameColumn).toTitleCase
    )))

  def heroes(): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table")
  }

  def search(name: String): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table WHERE $NameColumn LIKE '${name.toLowerCase}%'")
  }

  def fetchHero(id: String): ServiceCall[NotUsed, Hero] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    performRead(s"SELECT * FROM $Table WHERE $IdColumn = '$heroId'")
      .map(_.headOption.orNull)
```

```
  }
```

Since `session.selectAll` returns a `Future[Seq[Row]]`, we map the future and then map the sequence to transform rows into heroes. When fetching a single hero, we only look at the head of the sequence, returning null if no heroes are found. Another way would be to return `Option[Hero]` instead of `Hero`, thus avoiding nulls, but Lagom doesn't provide a serializer for options out of the box, and I have decided not to write a custom one for that. The HTTP response would look exactly the same if no hero is found, anyway.

Then, we implement the endpoints that call the write side.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
  private def entityRef(id: String): EntityRef[HeroCommand] =
    sharding.entityRefFor(HeroState.typeKey, id)

  def createHero(): ServiceCall[NewHero, Hero] = ServiceCall { hero =>
    val heroId = newId
    entityRef(heroId).ask[Confirmation](replyTo => CreateHero(hero.name, replyTo)).map
{
      case Accepted => Hero(heroId, hero.name.toTitleCase)
      case _ => throw BadRequest(s"Failed to create a hero with name ${hero.name}.")
    }
  }

  def changeHero(): ServiceCall[Hero, Done] = ServiceCall { hero =>
    val heroId = id2str(hero.id.toInt)
    entityRef(heroId).ask[Confirmation](replyTo => ChangeHero(hero.name, replyTo)).map
{
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to change a hero with id $heroId.")
    }
  }

  def deleteHero(id: String): ServiceCall[NotUsed, Done] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    entityRef(heroId).ask[Confirmation](replyTo => DeleteHero(replyTo)).map {
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to delete a hero with id $heroId.")
    }
  }
```

The `entityRef` method returns a handle through which to send messages to the underlying actors (note the use of `HeroState.typeKey` here). A message is sent by calling its `ask[Confirmation]` method, where `Confirmation` is the type of message we expect to receive in response. The underlying `ActorRef` is then passed to the command in its constructor argument: the `CreateHero` for the `createHero` endpoint, and so on.

The `ask` method returns a `Future[Confirmation]`, which becomes complete when a `Confirmation`

response is received or the timeout is reached (defined in the package object previously). We then map the future, throwing a `BadRequest` exception in case of rejection. Along the happy path, the newly created instance of `Hero` is returned from the `createHero` endpoint (to be send back to the frontend client, including the newly created hero's ID), while `Done` is reported by the others.

When creating a new hero, the underlying actor doesn't yet exist and gets created inside the `entityRef` method, where it is initialized with the empty state provided by `HeroState.initial`. That empty state is then immediately modified, of course, once the `CreateHero` command is processed.

13. Querying the History

The frontend client, provided by the Angular Tour of Heroes tutorial, does not have the functionality of querying for each hero's history—from creation through changes to possibly deletion—nor for the entire history of all heroes. However, for instructional purposes, we will give an example for how to query history in Lagom microservices.

First, an implicit instance of Akka `Materializer` must be injected into the `TourOfHeroesServiceImpl` class, in addition to all other injected arguments. Then, an instance of the read journal must be provided. Here, we use the write-side journal as a read side.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
class TourOfHeroesServiceImpl(sharding: ClusterSharding)
                             (readSide: ReadSide, cassandraReadSide:
CassandraReadSide, session: CassandraSession)
                             (implicit ec: ExecutionContext, materializer:
Materializer) extends TourOfHeroesService {
  ???

  private val readJournal = PersistenceQuery(materializer.system)
    .readJournalFor[CassandraReadJournal](CassandraReadJournal.Identifier)

  ???
}
```

We won't show here how to request the history of all heroes, nor how to request the IDs of all heroes ever created (and then possibly deleted), from which to select an individual hero. We will just show an example of a low-level query that can be used to implement a paged history for a single hero.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
  def history(id: String, from: Long, to: Long): ServiceCall[NotUsed, List[HeroEvent]]
= ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    val source = readJournal.eventsByPersistenceId(heroId, from, to)
    source.runFold(List.empty[HeroEvent]) { (acc, envelope) =>
      envelope.event match {
        case event: HeroEvent => event :: acc
        case _ => acc
      }
    }
  }
```

`eventsByPersistenceId` returns an Akka stream, so it would have been suited better to a streaming-style microservice, reporting to a WebSockets client. Here, we have to materialize the stream, for which the implicit materializer was needed.

In order to expose the event history as an HTTP endpoint, the `HeroEvent` trait would need to be moved to the `com.chariotsolutions.tohlagom.api` package. We won't do this, however. So, the `history` will remain as an example only, unused.

The first iteration of `TourOfHeroesServiceImpl.scala` now looks as follows, with all import statements included:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceImpl.scala

```
package com.chariotsolutions.tohlagom.impl

import akka.{Done, NotUsed}
import akka.stream.Materializer
import akka.persistence.query.PersistenceQuery
import scala.concurrent.{ExecutionContext, Future}
import akka.persistence.cassandra.query.scaladsl.CassandraReadJournal
import akka.cluster.sharding.typed.scaladsl.{ClusterSharding, EntityRef}
import com.lightbend.lagom.scaladsl.persistence.cassandra.{CassandraReadSide,
CassandraSession}
import com.lightbend.lagom.scaladsl.persistence.ReadSide
import com.lightbend.lagom.scaladsl.api.transport.BadRequest
import com.lightbend.lagom.scaladsl.api.ServiceCall
import com.chariotsolutions.tohlagom.api._

class TourOfHeroesServiceImpl(sharding: ClusterSharding)
                             (readSide: ReadSide, cassandraReadSide:
CassandraReadSide, session: CassandraSession)
                             (implicit ec: ExecutionContext, materializer:
Materializer) extends TourOfHeroesService {
  import HeroEventProcessor._
  readSide.register[HeroEvent](new HeroEventProcessor(cassandraReadSide, session))

  private def performRead(query: String): Future[Seq[Hero]] =
    session.selectAll(query).map(_.map(row => Hero(
      row.getString(IdColumn),
      row.getString(NameColumn).toTitleCase
    )))

  private def entityRef(id: String): EntityRef[HeroCommand] =
    sharding.entityRefFor(HeroState.typeKey, id)

  def heroes(): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table")
  }

  def search(name: String): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table WHERE $NameColumn LIKE '${name.toLowerCase}%'")
  }

  def fetchHero(id: String): ServiceCall[NotUsed, Hero] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
```

```
    performRead(s"SELECT * FROM $Table WHERE $IdColumn = '$heroId'")
      .map(_.headOption.orNull)
  }

  def createHero(): ServiceCall[NewHero, Hero] = ServiceCall { hero =>
    val heroId = newId
    entityRef(heroId).ask[Confirmation](replyTo => CreateHero(hero.name, replyTo)).map
{
      case Accepted => Hero(heroId, hero.name.toTitleCase)
      case _ => throw BadRequest(s"Failed to create a hero with name ${hero.name}.")
    }
  }

  def changeHero(): ServiceCall[Hero, Done] = ServiceCall { hero =>
    val heroId = id2str(hero.id.toInt)
    entityRef(heroId).ask[Confirmation](replyTo => ChangeHero(hero.name, replyTo)).map
{
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to change a hero with id $heroId.")
    }
  }

  def deleteHero(id: String): ServiceCall[NotUsed, Done] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    entityRef(heroId).ask[Confirmation](replyTo => DeleteHero(replyTo)).map {
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to delete a hero with id $heroId.")
    }
  }

  private val readJournal = PersistenceQuery(materializer.system)
    .readJournalFor[CassandraReadJournal](CassandraReadJournal.Identifier)

  def history(id: String, from: Long, to: Long): ServiceCall[NotUsed, List[HeroEvent]]
= ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    val source = readJournal.eventsByPersistenceId(heroId, from, to)
    source.runFold(List.empty[HeroEvent]) { (acc, envelope) =>
      envelope.event match {
        case event: HeroEvent => event :: acc
        case _ => acc
      }
    }
  }
}
```

14. Application Loader

It is now time to replace the template-generated file `TohlagomLoader.scala` with `TourOfHeroesLoader.scala`, where we define the application loader as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesLoader.scala

```
class TourOfHeroesLoader extends LagomApplicationLoader {
  override def describeService = Some(readDescriptor[TourOfHeroesService])

  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) {
      def serviceLocator: ServiceLocator = ServiceLocator.NoServiceLocator
    }

  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with LagomDevModeComponents
}
```

The `load` method currently provides no service locator. We will change this when going into production. For now, however, we are interested in making the microservice run in the development mode, which is what the `loadDevMode` method is for. The service locator for the development mode is provided by the `LoagomDevModeComponents` mixin.

Both `load` and `loadDevMode` methods return an instance of `TourOfHeroesApplication`, which extends `LagomApplication`. It is an abstract class, because it doesn't implement the `serviceLocator` method. Since we have decided to use Cassandra both for the write side and the read side, it will extend `CassandraPersistenceComponents`, among other things. Importantly, it must define `lagomServer`, as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesLoader.scala

```
abstract class TourOfHeroesApplication(context: LagomApplicationContext)
  extends LagomApplication(context) with CassandraPersistenceComponents with
AhcWSComponents {
  lazy val lagomServer = serverFor[TourOfHeroesService](wire[TourOfHeroesServiceImpl])
  clusterSharding.init(Entity(HeroState.typeKey)(HeroBehavior.create))

  ???
}
```

Here, `wire[TourOfHeroesServiceImpl]` is a Scala macro. Scala macros analyze the AST (Abstract Syntax Tree) produced by the Scala compiler and modify it according to the macro's purpose. In this case, `wire[TourOfHeroesServiceImpl]` finds any values in scope that match the types taken by the constructor arguments of `TourOfHeroesServiceImpl` and inject them into the constructor. Thus, the expression `wire[TourOfHeroesServiceImpl]` is replaced by the macros with an invocation of the `TourOfHeroesServiceImpl`'s constructor, where the arguments are provided by the various mixins included in the definition of `TourOfHeroesApplication`.

The `clusterSharding.init` call initializes sharding of the aggregates across the Akka cluster. In the development mode, an Akka cluster of one is automatically created, so we don't have to worry about joining it. But it will become important in the production mode.

Besides that, `TourOfHeroesApplication` also must implement a Json serializer registry. Below, we include all serializers we have defined so far, except for the commands, because they use Akka Jackson serialization instead of the Play Json module, as discussed before.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesLoader.scala

```
  lazy val jsonSerializerRegistry = new JsonSerializerRegistry {
    def serializers: List[JsonSerializer[_]] = List(
      JsonSerializer[HeroEvent],
      JsonSerializer[HeroCreated],
      JsonSerializer[HeroChanged],
      JsonSerializer[HeroDeleted.type],
      JsonSerializer[HeroState],
      JsonSerializer[Hero],
      JsonSerializer[NewHero],
      JsonSerializer[Confirmation],
      JsonSerializer[Accepted.type],
      JsonSerializer[Rejected]
    )
  }
```

The first iteration of the `TourOfHeroesLoader.scala` file now looks as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesLoader.scala

```
package com.chariotsolutions.tohlagom.impl

import com.softwaremill.macwire.wire
import play.api.libs.ws.ahc.AhcWSComponents
import akka.cluster.sharding.typed.scaladsl.Entity
import com.lightbend.lagom.scaladsl.api.ServiceLocator
import com.lightbend.lagom.scaladsl.playjson.{JsonSerializer, JsonSerializerRegistry}
import com.lightbend.lagom.scaladsl.server.{LagomApplication, LagomApplicationContext,
LagomApplicationLoader}
import
com.lightbend.lagom.scaladsl.persistence.cassandra.CassandraPersistenceComponents
import com.lightbend.lagom.scaladsl.devmode.LagomDevModeComponents
import com.chariotsolutions.tohlagom.api._

class TourOfHeroesLoader extends LagomApplicationLoader {
  override def describeService = Some(readDescriptor[TourOfHeroesService])

  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) {
      def serviceLocator: ServiceLocator = ServiceLocator.NoServiceLocator
    }
```

```
  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with LagomDevModeComponents
}

abstract class TourOfHeroesApplication(context: LagomApplicationContext)
  extends LagomApplication(context) with CassandraPersistenceComponents
    with AhcWSComponents with ClusterShardingTypedComponents {
  lazy val lagomServer = serverFor[TourOfHeroesService](wire[TourOfHeroesServiceImpl])
  clusterSharding.init(Entity(HeroState.typeKey)(HeroBehavior.create))

  lazy val jsonSerializerRegistry = new JsonSerializerRegistry {
    def serializers: List[JsonSerializer[_]] = List(
      JsonSerializer[HeroEvent],
      JsonSerializer[HeroCreated],
      JsonSerializer[HeroChanged],
      JsonSerializer[HeroDeleted.type],
      JsonSerializer[HeroState],
      JsonSerializer[Hero],
      JsonSerializer[NewHero],
      JsonSerializer[Confirmation],
      JsonSerializer[Accepted.type],
      JsonSerializer[Rejected]
    )
  }
}
```

15. Running in Development Mode

We are now almost ready to run our microservice in the development mode. We just need to configure `TourOfHeroesLoader` as the application loader and to define the Cassandra keyspace for the journal, snapshot store, and the read-side store. In addition to the Akka Jackson serialization bindings for `HeroCommand`, which we added earlier, the first iteration of our `application.conf` now looks as follows:

toh-lagom-impl/src/main/resources/application.conf

```
cassandra.default.keyspace = toh_lagom_dev
cassandra-journal.keyspace = ${cassandra.default.keyspace}
cassandra-snapshot-store.keyspace = ${cassandra.default.keyspace}
lagom.persistence.read-side.cassandra.keyspace = ${cassandra.default.keyspace}

akka.actor.serialization-bindings {
  "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
}

play.application.loader = com.chariotsolutions.tohlagom.impl.TourOfHeroesLoader
```

For instructional purposes, we didn't compile our code before. But now is the time to do this. With SBT, it is accomplished simply by running the following command in the same folder where `build.sbt` resides (that is, the `toh-lagom` folder at the top of our project hierarchy):

```
sbt clean compile
```

Here, the `clean` task is optional. You don't normally have to clean your `target` cache often. To run the microservice, execute the following command:

```
sbt runAll
```

In fact, the `runAll` task depends on the `compile` task, so the code will be compiled automatically if it wasn't already.

You will see some useful information reported in the standard output as the microservice starts up. Once the following line in green appears, the service is up, and its endpoints can be accessed on `localhost:9000`:

```
[info] (Service started, press enter to stop and go back to the console...)
```

Other useful messages include the notification about the embedded Cassandra and Kafka servers starting up. Since we don't use Kafka, it can be disabled by adding the following line to `build.sbt`, but it doesn't hurt to have it running:

build.sbt

```
lagomKafkaEnabled in ThisBuild := false
```

Changes made to the project itself, in `build.sbt` and any files in the `project` that it depends on, requires restarting the service. However, changes to the code of the microservice and any configuration files will be automatically picked up. The code will be recompiled (if necessary) and the service reloaded without you having to run `sbt runAll` again.

To run the service in the debug mode, use the following command instead:

```
sbt -jvm-debug 5005 runAll
```

The service will be listening for a debugger on the port 5005 (but you can specify a port of your choice instead). The service itself will be available on the port 9000. So, now you can send HTTP requests against `localhost:9000`, either using `curl` or via Postman, as we haven't hooked up the frontend client yet. For example:

```
curl -X POST -d '{"name": "Spiderman"}' http://localhost:9000/api/heroes
# ... wait 5 seconds (the default eventual consistency delay)
curl -X GET http://localhost:9000/api/heroes
```

Although it is possible to make a Lagom application serve the frontend assets (such as `index.html` and all the necessary Javascript and CSS files), we won't do this on purpose. The entire rationale behind the microservice architecture is to break up applications into loosely bound components, and integrating static frontend assets makes it anything *but* loosely bound. Once we deploy our microservice into AWS, we will use CloudFront to serve the static assets—not until then.

Before we move on to the production mode, however, let's say a few words about testing.

Part III. Testing

16. Unit Testing

There is an oft-repeated advice to write unit tests before implementing an application. This is a core principle of TDD (Test-Driven Development). In our case, when addressing a reader potentially unfamiliar with the Lagom framework or even with CQRS and Event Sourcing in general, the emphasis was mainly on showing first how things are put together. Now that we have done that, let's take a look at how to write unit tests for this. We won't aim for complete coverage but rather give an example of how to unit-test actors, which can be tricky. Thankfully, this is made easier by Akka's Actor Test Kit.

With SBT, as with Maven, unit test code is typically found under `project_dir/src/test/scala`.

The `logback.xml` file under `toh-lagom-impl/src/test/resources` serves to make the log output from the unit tests less verbose, since the default settings for unit test log output make it quite detailed.

Let's replace the template-generated `TohlagomAggregateSpec.scala` file with a new `HeroAggregateSpec.scala` file, in which we will write unit tests for the hero aggregate. Since it belongs to the `com.chariotsolutions.tohlagom.impl` package, it will have access to the `HeroBehavior.create` method that we have made private to this package. Although it is possible to use JUnit with Scala and SBT, we will continue using ScalaTest that the template-generated code used.

Here is how to define a class for unit-testing Lagom actors, while simulating the journal with an in-memory implementation and the snapshot store with temporary local storage.

ScalaTest users may be surprised by `AnyWordSpecLike`, but we are using a newer version of ScalaTest here, where `WordSpecLike` has been deprecated; we are also using `Matchers` from a different package.

toh-lagom-impl/src/test/scala/com/chariotsolutions/tohlagom/impl/HeroAggregateSpec.scala

```
class HeroAggregateSpec extends ScalaTestWithActorTestKit(s"""
      akka.persistence.journal.plugin = "akka.persistence.journal.inmem"
      akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
      akka.persistence.snapshot-store.local.dir = "target/snapshot-
${UUID.randomUUID().toString}"
    """) with AnyWordSpecLike with Matchers {
  "Hero aggregate" should {
    ???
  }
}
```

Let's take a look at one of the unit tests to see how it is organized.

toh-lagom-impl/src/test/scala/com/chariotsolutions/tohlagom/impl/HeroAggregateSpec.scala

```
    "create a hero with a given name" in {
```

```
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val stateProbe = createTestProbe[HeroState]()
      val name = "aLiCe"

      ref ! CreateHero(name, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! FetchHero(stateProbe.ref)
      stateProbe.expectMessage(HeroState(name.toLowerCase, valid = true))
    }
```

As you recall, `newId` is our helper function for generating new hero IDs. First, we combine it with a fake entity type to produce a `PersistenceId` and then pass it to our package-private `HeroBehavior.create` method to create an instance of `EventSourcedBehavior`. Once we pass it to the `spawn` function from the Actor Test Kit, an actor with that behavior is created but only the corresponding `ActorRef` is returned. Recall that instances of `Actor` must never be available to call methods on directly. Instead, we must send messages to an actor via its corresponding `ActorRef`.

Then, we use another useful function from the Actor Test Kit, `createTestProbe[T]`, to listen to all messages of the type `T` received by any actors in the current `ActorSystem`. Here, `confirmationProbe` listens for messages of the `Confirmation` type, and `stateProbe` listens for messages of the `HeroState` type.

When we send a `CreateHero` message to our actor—which is accomplished by calling the `!` method on the corresponding `ActorRef`--we expect to receive `Accepted` in response. This is tested by the `expectMessage` call. But this is not enough. We also must check that the actor's state has been properly updated. To do this, we send a `FetchHero` message to the same actor next, checking that the `HeroState` message sent back in response has the correct name and the `valid` flag. As you may recall, we have defined the `FetchHero` command for use in unit tests only, and this is how it is used.

Here is how the complete `HeroAggregateSpec.scala` file may look like:

toh-lagom-impl/src/test/scala/com/chariotsolutions/tohlagom/impl/HeroAggregateSpec.scala

```
package com.chariotsolutions.tohlagom.impl

import java.util.UUID
import scala.concurrent.duration._
import akka.persistence.typed.PersistenceId
import akka.actor.testkit.typed.scaladsl.ScalaTestWithActorTestKit
import org.scalatest.wordspec.AnyWordSpecLike
import org.scalatest.matchers.should.Matchers

class HeroAggregateSpec extends ScalaTestWithActorTestKit(s"""
      akka.persistence.journal.plugin = "akka.persistence.journal.inmem"
      akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
      akka.persistence.snapshot-store.local.dir = "target/snapshot-
${UUID.randomUUID().toString}"
    """) with AnyWordSpecLike with Matchers {
```

```
  "Hero aggregate" should {
    "create a hero with a given name" in {
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val stateProbe = createTestProbe[HeroState]()
      val name = "aLiCe"

      ref ! CreateHero(name, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! FetchHero(stateProbe.ref)
      stateProbe.expectMessage(HeroState(name.toLowerCase, valid = true))
    }

    "change an existing hero's name" in {
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val stateProbe = createTestProbe[HeroState]()
      val oldName = "aLiCe"
      val newName = "bOrIs"

      ref ! CreateHero(oldName, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! ChangeHero(newName, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! FetchHero(stateProbe.ref)
      stateProbe.expectMessage(HeroState(newName.toLowerCase, valid = true))
    }

    "fail to change a non-initialized hero" in {
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val newName = "bOrIs"

      ref ! ChangeHero(newName, confirmationProbe.ref)
      confirmationProbe.expectMessage(Rejected("The hero is in invalid state."))
    }

    "delete an existing hero" in {
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val stateProbe = createTestProbe[HeroState]()
      val name = "aLiCe"

      ref ! CreateHero(name, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! DeleteHero(confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)
```

```
      ref ! FetchHero(stateProbe.ref)
      stateProbe.expectNoMessage(timeout.duration)
    }

    "fail delete an already deleted hero" in {
      val ref = spawn(HeroBehavior.create(PersistenceId("fake-type-hint", newId)))
      val confirmationProbe = createTestProbe[Confirmation]()
      val name = "aLiCe"

      ref ! CreateHero(name, confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! DeleteHero(confirmationProbe.ref)
      confirmationProbe.expectMessage(Accepted)

      ref ! DeleteHero(confirmationProbe.ref)
      confirmationProbe.expectMessage(Rejected("The hero has already been deleted."))
    }
  }
}
```

These tests can be run by the following command:

```
sbt test
```

which is actually a shorthand for

```
sbt test:test
```

or

```
sbt test:testOnly your.test
```

to execute a single test.

However, you will need to delete, or comment out the contents of, the template-generated file `TohlagomServiceSpec.scala` first, for the `test:compile` task that `test:test` depends on to succeed. That file now references code that has been removed from other files, so it won't compile.

More importantly, that file actually contains not a unit test but an integration test, because it talks to Cassandra instead of mock actors as above. With SBT, there is a different place where integration tests properly belong, and we'll discuss it next.

17. Integration Testing

By default, in a Maven project, `project_dir/src/main` is reserved for the application code and `project_dir/src/test` for test code. In an SBT project, it's the same, except that the best practices call for placing only unit tests under `project_dir/src/test`, with integration tests placed under `project_dir/src/it`. The unit tests are executed by running

```
sbt test:test
```

(with `sbt test` as a shorthand for that), whereas integration tests are executed by running

```
sbt it:test
```

In addition, SBT allows defining an arbitrary number of custom test scopes, besides `Test` and `IntegrationTest` with their respective sets of `test:*` and `it:*` tasks (such as, for example, `it:clean`, `it:compile`, `it:test`, `it:testOnly`)--not necessarily by location in the code but, for example, by filtering the tests by name or applying any other programmatic criteria (SBT configuration is just code, after all). See here for more information on testing with SBT.

Whereas unit tests are typically fast, integration tests can take some time to run, because they may need to start and talk to databases and/or other microservices. When a typical programming workflow involves multiple pushes per day, it may not always be a good idea to execute integration tests automatically every time new code is pushed. A clear separation between unit and integration tests allows to configure the builds that automatically run upon a push to include only unit tests, whereas a nightly build may also execute all integration tests.

Just like `Test`, the `IntegrationTest` scope is predefined in SBT, but it is not enabled automatically. It must be explicitly added to the project. First, the following line must be added to the definition of the `toh-lagom-impl` subproject in `build.sbt`:

build.sbt

```
  .configs(IntegrationTest)
```

Second, the following must be added to the second, test-related, `.settings` call as an argument:

build.sbt

```
    Defaults.itSettings,
```

Third, ScalaTest must be made available not only in `Test` but also in `IntegrationTest`:

build.sbt

```
      scalaTest % Test,
```

```
      scalaTest % IntegrationTest,
```

But this is still not enough. It turns out that the `lagomScaladslTestKit` dependency is already scoped to `Test`. Instead of being defined in Lagom as `"com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"`, so that it could then be given different scopes depending on the need, it is actually defined as `"com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1" % Test`. So, unfortunately, we will have to duplicate some code and define it ourselves in `build.sbt`:

build.sbt

```
val lagomTestKit = "com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"
```

Then, we will replace `lagomScaladslTestKit` in the dependencies as follows (note that the test kit is not actually needed for unit tests):

build.sbt

```
      lagomTestKit % IntegrationTest
```

Finally, we also need to replace `lagomForkedTestSettings`, which are only defined for `Test`. These too are only needed for integration tests. In Lagom, all tests are *forked*, meaning that they are executed in separate JVMs that are forked from the initial one. Moreover, every test that uses Cassandra must run in its own JVM, not shared with other tests. This is accomplished by placing each test in its own `Tests.Group`, which is what `lagomForkedTestSettings` does under the hood, among other things.

While implementing similar settings for integration tests, we will do one better on Lagom by only placing Cassandra-related tests in single-test groups, whereas all other integration tests will be placed into one big group, because they can run in the same JVM. So, let's replace `lagomForkedTestSettings` with the following:

build.sbt

```
    TestSettings.forked(IntegrationTest),
```

in `build.sbt` and proceed to implementing the `TestSettings` object. The current iteration of `build.sbt` should look like this:

build.sbt

```
name in ThisBuild := "toh-lagom"
organization in ThisBuild := "com.chariotsolutions"
scalaVersion in ThisBuild := "2.13.1"
version in ThisBuild := "0.0.2"

scalacOptions in ThisBuild ++= Seq(
  "-language:implicitConversions",
  "-language:postfixOps"
)
```

```
lagomKafkaEnabled in ThisBuild := false

val lagomTestKit = "com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"

lazy val `toh-lagom` = (project in file("."))
  .aggregate(`toh-lagom-api`, `toh-lagom-impl`)

lazy val `toh-lagom-api` = (project in file("toh-lagom-api"))
  .settings(libraryDependencies ++= Seq(lagomScaladslApi))

lazy val `toh-lagom-impl` = (project in file("toh-lagom-impl"))
  .enablePlugins(LagomScala)
  .dependsOn(`toh-lagom-api`)
  .configs(IntegrationTest)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      macwire % Provided
    )
  )
  .settings(
    Defaults.itSettings,
    TestSettings.forked(IntegrationTest),
    libraryDependencies ++= Seq(
      scalaTest % Test,
      scalaTest % IntegrationTest,
      lagomTestKit % IntegrationTest
    )
  )
```

When describing the SBT project structure at the beginning of this tutorial, we mentioned that the `project` folder is for the code that `build.sbt` depends on. Now that we have added the TestSettings.forked(IntegrationTest) line to `build.sbt`, we need to implement it. Create a `TestSettings.scala` file in the `project` folder. Then, let me show what the end result will look like, after which we'll take a look at the code and see what it does.

project/TestSettings.scala

```
import sbt._
import sbt.Keys._
import xsbti.compile.CompileAnalysis
import xsbti.api.{Definition, Projection}

object TestSettings {
  def forked(scope: Configuration): Seq[Setting[_]] = Seq(
    fork in scope := true,
    javaOptions in scope ++= Seq(
```

```
      "-Xms256M", "-Xmx512M",
      "-Dconfig.file=toh-lagom-impl/src/it/resources/application.conf"
    ),
    testGrouping in scope := groupTests(
      (compile in scope).value,
      (definedTests in scope).value,
      (javaOptions in scope).value)
  )

  private def groupTests(analysis: CompileAnalysis, tests: Seq[TestDefinition],
                         javaOptions: Seq[String]): Seq[Tests.Group] = {
    val cassandraTestNames: Set[String] = Tests.allDefs(analysis)
      .withFilter(hasAnnotation("RequiresCassandra")).map(_.name).toSet
    val (cassandraTests, otherTests) = tests.partition(test => cassandraTestNames
.contains(test.name))
    val runPolicy = Tests.SubProcess(ForkOptions().withRunJVMOptions(javaOptions
.toVector))
    val cassandraTestGroups = cassandraTests.map(test => Tests.Group(test.name, Seq
(test), runPolicy))
    val otherTestsGroup = Tests.Group("Other tests", otherTests, runPolicy)
    otherTestsGroup +: cassandraTestGroups
  }

  def hasAnnotation(annotationId: String): Definition => Boolean =
    definition => definition.annotations.exists(_.base match {
      case proj: Projection => proj.id == annotationId
      case _ => false
    })
}
```

The first thing to note is that we pass `scope` as an argument. Even though we will only invoke that method for `IntegrationTest`, it may come handy in the future if some custom test scopes must be added that require forking tests. Then, we set `fork` to `true` and `javaOptions` for the forked JVM to reasonable memory settings. We also specify an `application.conf` file that is placed under `toh-lagom-impl/src/it/resources` instead of `toh-lagom-impl/src/main/resources`. For now, let's copy `application.conf` from the latter location to the former. But in the future, they will diverge. Also, copy `logback.xml` from `toh-lagom-impl/src/test/resources` to `toh-lagom-impl/src/it/resources`, in order to make the standard output less verbose.

Then, we define `testGrouping` in the scope by calling a helper function `groupTests`, passing to it the instance of `CompileAnalysis` from the `scope:compile` task (which has information about the code in the test files), the list of tests defined in the scope, and the JVM options.

`Tests.allDefs(analysis)` gives us all test suits, which we filter by having a `@RequiresCassandra` annotation, and obtains a set of their names (because these are instances of `xsbti.api.Definition`, not of `sbt.TestDefinition`, with which we must compare them). Then, we partition the defined tests (instances of `sbt.TestDefinition`) based on whether their names are in the set of names of Cassandra tests or not. For each Cassandra test, we form a group with that test as a single member. All other tests are made into a single group, which is prepended to the sequence of Cassandra test

groups.

This demonstrates the power and flexibility of SBT, compared to XML-based build tools such as Maven. Instead of waiting for the next release of a third-party Maven plugin—which may or may not address the specific requirements of your use case—you can write actual code as part of the build configuration, the code geared specifically to your scenario.

Now that we have configured the `IntegrationTest` scope for our project properly, let's write some tests.

18. Writing Integration Tests

If you haven't already done so, create a `toh-lagom-impl/src/it` folder for integration tests and, under it, three subfolders: `resources`, `java`, and `scala` (yes, we'll need some Java for this). Into `resources`, copy `application.conf` from `toh-lagom-impl/src/main/resources` (these configuration files will soon diverge in content) and `logback.xml` from `toh-lagom-impl/src/test/resources` (to make the standard output from tests less verbose).

Because of certain differences between Java annotations and Scala annotations, we'll have to use a Java annotation to mark Cassandra tests. Actually, Scala and Java files can mix, all residing under the `scala` folder. But in this case, we're separating not just Java from Scala but annotations from tests. So, under the `java` folder, create the `com.chariotsolutions.tohlagom.impl` project and, under it, the `@RequiresCassandra` annotation.

toh-lagom-impl/src/it/java/com/chariotsolutions/tohlagom/impl/RequiresCassandra.java

```
package com.chariotsolutions.tohlagom.impl;

import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface RequiresCassandra {
}
```

Then, create the same package under `scala` and, in that package, the following test suite. We don't aim for complete test coverage but rather want to give an example of how to write them. It is entirely possible to add more tests in the future—and even more annotations, with different test grouping behaviors, such as `@RequiresCluster` or `@RequiresJDBC`, for example.

toh-lagom-impl/src/it/scala/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceSpec.scala

```
package com.chariotsolutions.tohlagom.impl

import akka.Done
import scala.concurrent.duration._
import org.scalatest.BeforeAndAfterAll
import org.scalatest.wordspec.AsyncWordSpec
import org.scalatest.matchers.should.Matchers
import com.lightbend.lagom.scaladsl.testkit.ServiceTest
import com.lightbend.lagom.scaladsl.server.LocalServiceLocator
import com.chariotsolutions.tohlagom.api._

@RequiresCassandra
class TourOfHeroesServiceSpec extends AsyncWordSpec with Matchers with
```

```
BeforeAndAfterAll {
  private val server: ServiceTest.TestServer[_ <: TourOfHeroesApplication] =
ServiceTest.startServer(ServiceTest.defaultSetup.withCassandra) { ctx =>
      new TourOfHeroesApplication(ctx) with LocalServiceLocator
    }

  val client = server.serviceClient.implement[TourOfHeroesService]
  override protected def afterAll() = server.stop()
  val pause = 30 seconds

  "toh-lagom service" should {
    val name1 = "aLiCe"
    val name2 = "bOrIs"
    val name3 = "aNnA"
    val newName = "cOrInNe"
    val prefix = "A"

    "create heroes" in {
      for {
        hero1 <- client.createHero().invoke(NewHero(name1))
        hero2 <- client.createHero().invoke(NewHero(name2))
        hero3 <- client.createHero().invoke(NewHero(name3)) } yield { hero1.name
should ===(name1.toTitleCase) hero2.name should ===(name2.toTitleCase) hero3.name
should ===(name3.toTitleCase) } } "fetch all heroes" in { Thread.sleep(pause.toMillis)
client.heroes().invoke().map { heroes =>
        heroes.size should ===(3)
        val match1 = heroes.find(_.name == name1.toTitleCase)
        val match2 = heroes.find(_.name == name2.toTitleCase)
        val match3 = heroes.find(_.name == name3.toTitleCase)
        match1.isDefined should ===(true)
        match2.isDefined should ===(true)
        match3.isDefined should ===(true)
      }
    }

    "search heroes by prefix" in {
      client.search(prefix).invoke().map { heroes =>
        heroes.foreach(_.name.substring(0, prefix.length).toTitleCase should
===(prefix.toTitleCase))
        heroes.size should ===(2)
      }
    }

    "fetch an existing hero" in {
      for {
        heroes <- client.heroes().invoke()
        heroId = heroes.head.id
        heroName = heroes.head.name
        hero <- client.fetchHero(heroId).invoke()
      } yield {
        hero.id should ===(heroId)
```

```
        hero.name should ===(heroName.toTitleCase)
      }
    }

    "change a hero's name" in {
      for {
        heroes <- client.heroes().invoke()
        heroId = heroes.head.id
        answer <- client.changeHero().invoke(Hero(heroId, newName))
        _ = Thread.sleep(pause.toMillis)
        hero <- client.fetchHero(heroId).invoke()
      } yield {
        answer should ===(Done)
        hero.id should ===(heroId)
        hero.name should ===(newName.toTitleCase)
      }
    }

    "delete a hero" in {
      for {
        heroes <- client.heroes().invoke()
        heroId = heroes.head.id
        answer <- client.deleteHero(heroId).invoke()
        _ = Thread.sleep(pause.toMillis)
        remaining <- client.heroes().invoke()
      } yield {
        answer should ===(Done)
        remaining.size should ===(2)
        remaining.map(_.id).contains(heroId) should ===(false)
      }
    }
  }
}
```

An important thing here is the `.withCassandra` call when defining a test server. It allows us using Cassandra even when the embedded Cassandra server is disabled in `build.sbt` (as it will be, soon). Also, note the use of `LocalServiceLocator` as the service locator for integration tests and `@RequiresCassandra` annotating the entire suite.

The test cases are self-explanatory. First, we create some heroes, then test the other endpoints on them. The pause accounts for the delay between processing a command by the write side and the processing the corresponding events by the read side.

Next, we'll make the first step toward making our Tour of Heroes microservice work in the production mode by replacing the embedded Cassandra server with AWS Managed Cassandra Service.

Part IV. Mixed Persistence

19. Switching to AWS Cassandra

Amazon Keyspaces (formerly known as AWS Managed Cassandra Service) is a new AWS service that debuted in December 2019 in the preview mode and became generally available in April 2020. Amazon Keyspaces does have some functional differences from Apache Cassandra but is very similar. In this document, we will also colloquially call it AWS Cassandra.

Unlike most AWS services, Amazon Keyspaces is accessed programmatically not via AWS SDK but with most Cassandra drivers that currently exist, in pretty much the same way as for Apache Cassandra, with some differences that we are about to describe. Accordingly, your AWS access keys won't work for this. Instead, you must generate server-specific credentials for Amazon Keyspaces, for which there is a special section in the **Security Credentials** tab for users in the AWS IAM console. These are username-password credentials. For information on how to generate them via the AWS console or programmatically, see here.

From now on, we will assume that you have an AWS user with appropriate permissions and with Amazon Keyspaces service-specific credentials generated. These need to be added to the application's configuration file in the manner describe below. However, in order to separate Cassandra-specific configuration settings from others, let's create a `cassandra.conf` file under `toh-lagom-impl/src/main/resources`, as follows:

toh-lagom-impl/src/main/resources/cassandra.conf

```
cassandra.default {
  keyspace = toh_lagom_dev
  read-consistency = "LOCAL_QUORUM"
  write-consistency = "LOCAL_QUORUM"
  replication-factor = 3

  ssl.truststore {
    path = "cassandra_truststore.jks"
    password = "amazon"
  }

  authentication {
    username = ${AWS_MCS_USERNAME}
    password = ${AWS_MCS_PASSWORD}
  }
}

cassandra-journal {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = ${cassandra.default.read-consistency}
  write-consistency = ${cassandra.default.write-consistency}
  replication-factor = ${cassandra.default.replication-factor}

  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }
```

```
  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
  }
}

cassandra-query-journal {
  read-consistency = ${cassandra.default.read-consistency}
}

cassandra-snapshot-store {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = "ONE"
  write-consistency = "ONE"
  replication-factor = ${cassandra.default.replication-factor}

  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }

  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
  }
}

lagom.persistence.read-side.cassandra {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = ${cassandra.default.read-consistency}
  write-consistency = ${cassandra.default.write-consistency}
  replication-factor = ${cassandra.default.replication-factor}

  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }

  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
  }
}
```

Here, `cassandra-query-journal` refers to the read journal used to retrieve the event history. The `replication-factor` is currently ignored by AWS, which always sets it to 3. But it may become important later on, as AWS Cassandra matures. Still, we'll remove it later, when disabling auto-create of the Cassandra keyspace for production.

We have moved the keyspace settings out of `application.conf`, too. Now, `application.conf` looks as follows:

toh-lagom-impl/src/main/resources/application.conf

```
include "cassandra"

akka.actor.serialization-bindings {
  "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
}

play.application.loader = com.chariotsolutions.tohlagom.impl.TourOfHeroesLoader
```

By default, the read and write consistency in the embedded Cassandra server is set to `QUORUM`, except for the snapshot store, where it is set to `ONE`. However, AWS Cassandra doesn't support `QUORUM`, and the AWS MCS Developer's Guide suggests using `LOCAL_QUORUM` instead.

Instead of committing the service-specific credentials into source code, we reference them via the `AWS_MCS_USERNAME` and `AWS_MCS_PASSWORD` environment variables. The question mark means that if the environment variable isn't defined, the previous value (if any) is to be used. Make sure to update the script you use to set up the AWS environment for your use (such as `AWS_REGION`, `AWS_ACCESS_KEY_ID`, and `AWS_SECRET_ACCESS_KEY`) with these values, too.

Besides the username and password for authentication, `cassandra.conf` also specifies a trust store for SSL. It is obtained by following the AWS-provided instructions. First, we download the Amazon digital certificate to a folder of your choice:

```
curl https://www.amazontrust.com/repository/AmazonRootCA1.pem -O
```

Then, we convert it to a trust store file:

```
openssl x509 -outform der -in AmazonRootCA1.pem -out temp_file.der
keytool -import -alias cassandra -keystore cassandra_truststore.jks -file
temp_file.der
```

Make sure to provide "amazon" as password. Then, we remove the temporary files, no longer needed, and move `cassandra_truststore.jks` to the root of our project, the `toh-lagom` folder. It is to be specified in `cassandra.conf` with "amazon" as password, as shown above.

Now, we must disable the embedded Cassandra server and point our microservice at an AWS Cassandra endpoint. The Amazon Keyspaces service is managed, so any IP addresses to which the endpoints resolve at any given time aren't going to last. The AWS Cassandra endpoints have the following form: `https://cassandra.us-east-1.amazonaws.com:9142`, where `us-east-1` can be replaced by another region (not all regions are supported at the moment, though the coverage is sure to expand).

At this time, we don't yet implement the production mode in our Lagom microservice. We are going

one small step after another. Our intent is to replace the embedded Cassandra server with AWS Cassandra in the development mode, and this is done by adding the following to `build.sbt` (but note that this only applies to the development mode and will be done differently for the production mode):

build.sbt

```
import scala.util.Try
import com.amazonaws.regions.{Region, Regions}

lagomCassandraEnabled in ThisBuild := false
lagomUnmanagedServices in ThisBuild := Map(
  "cas_native" -> s"""https://cassandra.${sys.env("AWS_REGION")}.amazonaws.com:9142"""
)
```

When running the microservice in the development mode, one must now login first to AWS. The current AWS region will then be used, as defined in the `AWS_REGION` environment variable by the AWS initialization script.

The triple quotes allow us not to escape the quotes within the string.

In order to be able to use the `com.amazonaws.regions` package, the AWS Java SDK must be added to the list of dependencies used by `build.sbt`. This is not the same as adding dependencies to the `toh-lagom` project. Recall that the code called from `build.sbt` resides in the `project` folder. Since SBT is recursive, adding dependencies to the build code itself is accomplished by creating a `project/build.sbt` file with the following line in it:

project/build.sbt

```
libraryDependencies += "com.amazonaws" % "aws-java-sdk" % "1.11.762"
```

as well as set the SBT version in `project/project/build.properties`. After this, the main project's `build.sbt` file will compile and look as follows:

build.sbt

```
import scala.util.Try
import com.amazonaws.regions.{Region, Regions}

name in ThisBuild := "toh-lagom"
organization in ThisBuild := "com.chariotsolutions"
scalaVersion in ThisBuild := "2.13.1"
version in ThisBuild := "0.1.0"

scalacOptions in ThisBuild ++= Seq(
  "-language:implicitConversions",
  "-language:postfixOps"
)
```

```
lagomKafkaEnabled in ThisBuild := false
lagomCassandraEnabled in ThisBuild := false
lagomUnmanagedServices in ThisBuild := Map(
  "cas_native" -> s"""https://cassandra.${sys.env("AWS_REGION")}.amazonaws.com:9142"""
)

val lagomTestKit = "com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"

lazy val `toh-lagom` = (project in file("."))
  .aggregate(`toh-lagom-api`, `toh-lagom-impl`)

lazy val `toh-lagom-api` = (project in file("toh-lagom-api"))
  .settings(libraryDependencies ++= Seq(lagomScaladslApi))

lazy val `toh-lagom-impl` = (project in file("toh-lagom-impl"))
  .enablePlugins(LagomScala)
  .dependsOn(`toh-lagom-api`)
  .configs(IntegrationTest)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      macwire % Provided
    )
  )
  .settings(
    Defaults.itSettings,
    TestSettings.forked(IntegrationTest),
    libraryDependencies ++= Seq(
      scalaTest % Test,
      scalaTest % IntegrationTest,
      lagomTestKit % IntegrationTest
    )
  )
```

20. Running with AWS Cassandra

When starting our microservice in the development mode (with `sbt runAll`, as before), the `toh_lagom` keyspace and the necessary tables will be automatically created—although the first requests may result in errors until that process is complete. However, you will see the following error:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Unsupported
statement type class org.apache.cassandra.cql3.statements.CreateIndexStatement
```

This happens when the event processor create the `hero` table from its global prepare callback and then attempts to create an index for it (see `HeroEventProcessor.scala`). It turns out that AWS Cassandra doesn't supported indexes, at least at this time.

Unfortunately, even if we sacrifice on the search-by-prefix functionality, replacing it with exact match and removing the index statement from the code, the problems don't end there. When trying to create a new hero (with `curl` or Postman), you may or may not see the following error:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Empty string is
not yet supported
```

(though, hopefully, it would be supported by the time you read this). The empty string is the default value for the `evebt_manifest` column of the `messages` and `tag_views` tables in the journal. To work around this, we will need to replace the manifest with, say, an empty space. In `HeroBehavior.scala`, add the following code, which is no-op in everything except for setting the manifest:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroBehavior.scala

```
object HeroEventAdapter extends EventAdapter[HeroEvent, HeroEvent] {
  def manifest(event: HeroEvent): String = " "
  def fromJournal(event: HeroEvent, manifest: String) = EventSeq.single(event)
  def toJournal(event: HeroEvent) = event
}
```

Then, in `HeroBehavior.create` (the public method), add a `.eventAdapter(HeroEventAdapter)` call, as follows:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/HeroBehavior.scala

```
  def create(entityContext: EntityContext[HeroCommand]): Behavior[HeroCommand] = {
    val persistenceId = PersistenceId(entityContext.entityTypeKey.name, entityContext
.entityId)
    create(persistenceId).eventAdapter(HeroEventAdapter).withTagger(
      AkkaTaggerAdapter.fromLagom(entityContext, HeroEvent.Tag)
    )
  }
```

Although this is certainly a workaround, it is still instructive to see how one can add event adapters that modify events before saving them in the journal. One can, for example, add custom tags to each event, which can then be used to filter the event history.

Once that is done, we will start seeing event saved in the AWS Cassandra's `messages` table (which is how the event journal is implemented internally in the Cassandra plugin) in the `toh_lagom` keyspace (you can use the AWS console to browse them). However, nothing still appears in the `hero` table, representing the read side. What's going on?

Looking at the standard output, you'll notice the following error:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Only UNLOGGED
Batches are supported at this time.
```

Recall that the event handlers in `HeroEventProcessor` return lists of statements to be executed atomically, together with the update to the offset table, as a logged batch by the Lagom framework. Unfortunately, AWS Cassandra doesn't support logged batches, at least at the time of this writing. There is no good workaround for this. Even if we execute the statements in the event handlers, the update to the offset table will still fail, so Lagom will attempt to process the same events all over again—and since it does this by small groups of events, we will never proceed beyond only a few. Even if we duplicate the offset-updating code from Lagom, this will be ugly and there will still be exceptions in the standard output about this, because Lagom will continue to attempt the updates.

So, we won't proceed down this path. Instead, let's use this as a teaching moment to show how to implement a Lagom microservice that uses different databases as its write and read sides. We will continue using AWS Cassandra as the write side (after the manifest fix, it works beautifully in this capacity), but we will use a PostgreSQL database—eventually, AWS Aurora in the PostgreSQL mode—as the read side.

21. Lagom in Mixed Persistence Mode

First, we must add the JDBC dependencies to `build.sbt`:

build.sbt

```
import scala.util.Try
import com.amazonaws.regions.{Region, Regions}

name in ThisBuild := "toh-lagom"
scalaVersion in ThisBuild := "2.13.1"
organization in ThisBuild := "com.chariotsolutions"
version in ThisBuild := "0.1.0"

scalacOptions in ThisBuild ++= Seq(
  "-language:implicitConversions",
  "-language:postfixOps"
)

lagomKafkaEnabled in ThisBuild := false
lagomCassandraEnabled in ThisBuild := false
lagomUnmanagedServices in ThisBuild := Map(
  "cas_native" -> s"""https://cassandra.${sys.env("AWS_REGION")}.amazonaws.com:9142"""
)

val lagomTestKit = "com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
val postgresql = "org.postgresql" % "postgresql" % "42.2.10"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"

lazy val `toh-lagom` = (project in file("."))
  .aggregate(`toh-lagom-api`, `toh-lagom-impl`)

lazy val `toh-lagom-api` = (project in file("toh-lagom-api"))
  .settings(libraryDependencies ++= Seq(lagomScaladslApi))

lazy val `toh-lagom-impl` = (project in file("toh-lagom-impl"))
  .enablePlugins(LagomScala)
  .dependsOn(`toh-lagom-api`)
  .configs(IntegrationTest)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      lagomScaladslPersistenceJdbc,
      postgresql % Runtime,
      macwire % Provided
    )
  )
  .settings(
    Defaults.itSettings,
```

```
    TestSettings.forked(IntegrationTest),
    libraryDependencies ++= Seq(
      scalaTest % Test,
      scalaTest % IntegrationTest,
      lagomTestKit % IntegrationTest
    )
  )
```

Here, `postgresql` is the PostgreSQL JDBC driver dependency, while `lagomScaladslPersistenceJdbc` is Lagom's persistence library for JDBC.

While refactoring our microservice to work with Cassandra as the write side and a PostgreSQL database as the read side, we will place the new, mixed-mode code into the `com.chariotsolutions.tohlagom.mixed` package, while keeping the Cassandra-only code in the `com.chariotsolutions.tohlagom.cassandra` package—in the hope that, in due time, the necessary functionality will be implemented in AWS Cassandra, at which time both versions of our microservice would be useable. Moreover, we will keep our integration tests running against Cassandra as both sides, for which we'll need Cassandra-only code, anyway. Adding integration tests for the mixed mode will be left as an exercise.

Another new package, `com.chariotsolutions.tohlagom.common` will contain the base traits and classes for both Cassandra-only and mixed mode, with the methods that are common to both. Let's begin by moving `TourOfHeroesLoader.scala` from `com.chariotsolutions.tohlagom.impl` to `com.chariotsolutions.tohlagom.common` and leaving only the following:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/common/TourOfHeroesLoader.scala

```
package com.chariotsolutions.tohlagom.common

import play.api.libs.ws.ahc.AhcWSComponents
import akka.cluster.sharding.typed.scaladsl.Entity
import com.lightbend.lagom.scaladsl.server.LagomApplicationLoader
import com.lightbend.lagom.scaladsl.cluster.typed.ClusterShardingTypedComponents
import com.lightbend.lagom.scaladsl.playjson.{JsonSerializer, JsonSerializerRegistry}
import com.chariotsolutions.tohlagom.impl._
import com.chariotsolutions.tohlagom.api._

trait TourOfHeroesLoader extends LagomApplicationLoader {
  override def describeService = Some(readDescriptor[TourOfHeroesService])
}

trait TourOfHeroesApplication extends AhcWSComponents with
ClusterShardingTypedComponents {
  clusterSharding.init(Entity(HeroState.typeKey)(HeroBehavior.create))

  lazy val jsonSerializerRegistry = new JsonSerializerRegistry {
    def serializers: List[JsonSerializer[_]] = List(
      JsonSerializer[HeroEvent],
      JsonSerializer[HeroCreated],
      JsonSerializer[HeroChanged],
```

```
      JsonSerializer[HeroDeleted.type],
      JsonSerializer[HeroState],
      JsonSerializer[Hero],
      JsonSerializer[NewHero],
      JsonSerializer[Confirmation],
      JsonSerializer[Accepted.type],
      JsonSerializer[Rejected]
    )
  }
}
```

Here, we need the `ClusterShardingTypedComponents` mixin to be specified explicitly, whereas it used to be included with `CassandraPersistenceComponents`.

In `com.chariotsolutions.tohlagom.cassandra`, create a `TourOfHeroesLoader.scala` file, into which we'll move everything else that didn't go into the base traits above:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/cassandra/TourOfHeroesLoader.scala

```
package com.chariotsolutions.tohlagom.cassandra

import com.softwaremill.macwire.wire
import com.lightbend.lagom.scaladsl.devmode.LagomDevModeComponents
import com.lightbend.lagom.scaladsl.akka.discovery.AkkaDiscoveryComponents
import com.lightbend.lagom.scaladsl.server.{LagomApplication, LagomApplicationContext}
import
com.lightbend.lagom.scaladsl.persistence.cassandra.CassandraPersistenceComponents
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.api._

class TourOfHeroesLoader extends common.TourOfHeroesLoader {
  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) {
      def serviceLocator: ServiceLocator = ServiceLocator.NoServiceLocator
    }

  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with LagomDevModeComponents
}

abstract class TourOfHeroesApplication(context: LagomApplicationContext)
  extends LagomApplication(context) with common.TourOfHeroesApplication with
CassandraPersistenceComponents {
  lazy val lagomServer = serverFor[TourOfHeroesService](wire[TourOfHeroesServiceImpl])
}
```

The corresponding `TourOfHeroesLoader.scala` file in `com.chariotsolutions.tohlagom.mixed` will look similar, except that instead of mixing in `CassandraPersistenceComponents` into `TourOfHeroesApplication`, we will mixed in `WriteSideCassandraPersistenceComponents` for the write

side, `ReadSideJdbcPersistenceComponents` for the read side, and `HikariCPComponents` for JDBC connection management.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/TourOfHeroesLoader.scala

```
package com.chariotsolutions.tohlagom.mixed

import com.softwaremill.macwire.wire
import com.lightbend.lagom.scaladsl.devmode.LagomDevModeComponents
import com.lightbend.lagom.scaladsl.akka.discovery.AkkaDiscoveryComponents
import com.lightbend.lagom.scaladsl.server.{LagomApplication, LagomApplicationContext}
import
com.lightbend.lagom.scaladsl.persistence.cassandra.CassandraPersistenceComponents
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.api._

class TourOfHeroesLoader extends common.TourOfHeroesLoader {
  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) {
      def serviceLocator: ServiceLocator = ServiceLocator.NoServiceLocator
    }

  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with LagomDevModeComponents
}

abstract class TourOfHeroesApplication(context: LagomApplicationContext)
  extends LagomApplication(context) with common.TourOfHeroesApplication
    with ReadSideJdbcPersistenceComponents with HikariCPComponents
    with WriteSideCassandraPersistenceComponents {
  lazy val lagomServer = serverFor[TourOfHeroesService](wire[TourOfHeroesServiceImpl])
}
```

Move `TourOfHeroesServiceImpl.scala` from `com.chariotsolutions.tohlagom.impl` into `com.chariotsolutions.tohlagom.common`. We will make the helper method `performRead` abstract, with different implementations for Cassandra and PostgreSQL—as it happens, the CQL and SQL syntax for the statements is identical but the driver calls are different. We will also factor out the event processor registered with the read side into an abstract method `eventProcessor`. We will also remove the Cassandra-specific read-side constructor arguments.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/common/TourOfHeroesServiceImpl.scala

```
package com.chariotsolutions.tohlagom.common

import akka.{Done, NotUsed}
import akka.stream.Materializer
import akka.persistence.query.PersistenceQuery
import scala.concurrent.{ExecutionContext, Future}
import akka.persistence.cassandra.query.scaladsl.CassandraReadJournal
import akka.cluster.sharding.typed.scaladsl.{ClusterSharding, EntityRef}
```

```
import com.lightbend.lagom.scaladsl.persistence.{ReadSide, ReadSideProcessor}
import com.lightbend.lagom.scaladsl.api.transport.BadRequest
import com.lightbend.lagom.scaladsl.api.ServiceCall
import com.chariotsolutions.tohlagom.impl._
import com.chariotsolutions.tohlagom.api._

abstract class TourOfHeroesServiceImpl(sharding: ClusterSharding)(readSide: ReadSide)
                                      (implicit ec: ExecutionContext, materializer:
Materializer) extends TourOfHeroesService {
  import HeroEventProcessor._
  readSide.register[HeroEvent](eventProcessor)
  protected def eventProcessor: ReadSideProcessor[HeroEvent]
  protected def performRead(query: String): Future[Seq[Hero]]

  private def entityRef(id: String): EntityRef[HeroCommand] =
    sharding.entityRefFor(HeroState.typeKey, id)

  def heroes(): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table")
  }

  def search(name: String): ServiceCall[NotUsed, Seq[Hero]] = ServiceCall { _ =>
    performRead(s"SELECT * FROM $Table WHERE $NameColumn LIKE '${name.toLowerCase}%'")
  }

  def fetchHero(id: String): ServiceCall[NotUsed, Hero] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    performRead(s"SELECT * FROM $Table WHERE $IdColumn = '$heroId'")
      .map(_.headOption.orNull)
  }

  def createHero(): ServiceCall[NewHero, Hero] = ServiceCall { hero =>
    val heroId = newId
    entityRef(heroId).ask[Confirmation](replyTo => CreateHero(hero.name, replyTo)).map
{
      case Accepted => Hero(heroId, hero.name.toTitleCase)
      case _ => throw BadRequest(s"Failed to create a hero with name ${hero.name}.")
    }
  }

  def changeHero(): ServiceCall[Hero, Done] = ServiceCall { hero =>
    val heroId = id2str(hero.id.toInt)
    entityRef(heroId).ask[Confirmation](replyTo => ChangeHero(hero.name, replyTo)).map
{
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to change a hero with id $heroId.")
    }
  }

  def deleteHero(id: String): ServiceCall[NotUsed, Done] = ServiceCall { _ =>
    val heroId = id2str(id.toInt)
```

```
    entityRef(heroId).ask[Confirmation](replyTo => DeleteHero(replyTo)).map {
      case Accepted => Done
      case _ => throw BadRequest(s"Failed to delete a hero with id $heroId.")
    }
  }

  private val readJournal = PersistenceQuery(materializer.system)
    .readJournalFor[CassandraReadJournal](CassandraReadJournal.Identifier)

  def history(id: String, from: Long, to: Long): ServiceCall[NotUsed, List[HeroEvent]]
= ServiceCall { _ =>
    val heroId = id2str(id.toInt)
    val source = readJournal.eventsByPersistenceId(heroId, from, to)
    source.runFold(List.empty[HeroEvent]) { (acc, envelope) =>
      envelope.event match {
        case event: HeroEvent => event :: acc
        case _ => acc
      }
    }
  }
}
```

Then, the Cassandra-only extension of this base class needs to implement those two abstract methods, which we copy from the implementation we had before.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/cassandra/TourOfHeroesServiceImpl.scala

```
package com.chariotsolutions.tohlagom.cassandra

import akka.stream.Materializer
import scala.concurrent.{ExecutionContext, Future}
import akka.cluster.sharding.typed.scaladsl.ClusterSharding
import com.lightbend.lagom.scaladsl.persistence.cassandra.{CassandraReadSide,
CassandraSession}
import com.lightbend.lagom.scaladsl.persistence.ReadSide
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.impl._
import com.chariotsolutions.tohlagom.api._
import common.HeroEventProcessor._

class TourOfHeroesServiceImpl(sharding: ClusterSharding)
                             (readSide: ReadSide, cassandraReadSide:
CassandraReadSide, session: CassandraSession)
                             (implicit ec: ExecutionContext, materializer:
Materializer)
  extends common.TourOfHeroesServiceImpl(sharding)(readSide) {
  protected lazy val eventProcessor = new HeroEventProcessor(cassandraReadSide,
session)

  protected def performRead(query: String): Future[Seq[Hero]] =
```

```
    session.selectAll(query).map(_.map(row => Hero(
      row.getString(IdColumn),
      row.getString(NameColumn).toTitleCase
    )))
}
```

In `com.chariotsolutions.tohlagom.mixed`, we will inject instances of `JdbcReadSide` and `JdbcSession` instead, and then implement the abstract methods as follows.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/TourOfHeroesServiceImpl.scala

```
package com.chariotsolutions.tohlagom.mixed

import akka.stream.Materializer
import scala.collection.immutable.VectorBuilder
import scala.concurrent.{ExecutionContext, Future}
import akka.cluster.sharding.typed.scaladsl.ClusterSharding
import com.lightbend.lagom.scaladsl.persistence.jdbc.{JdbcReadSide, JdbcSession}
import com.lightbend.lagom.scaladsl.persistence.ReadSide
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.impl._
import com.chariotsolutions.tohlagom.api._
import common.HeroEventProcessor._

class TourOfHeroesServiceImpl(sharding: ClusterSharding)
                              (readSide: ReadSide, jdbcReadSide: JdbcReadSide, session:
JdbcSession)
                              (implicit ec: ExecutionContext, materializer:
Materializer)
  extends common.TourOfHeroesServiceImpl(sharding)(readSide) {
  protected lazy val eventProcessor = new HeroEventProcessor(jdbcReadSide)

  protected def performRead(query: String): Future[Seq[Hero]] =
    session.withConnection { connection =>
      JdbcSession.tryWith(connection.prepareStatement(query)) { stmt =>
        JdbcSession.tryWith(stmt.executeQuery()) { row =>
          val summaries = new VectorBuilder[Hero]
          while (row.next()) {
            summaries += Hero(
              row.getString(IdColumn),
              row.getString(NameColumn).toTitleCase
            )
          }
          summaries.result()
        }
      }
    }
}
```

Now, we only need to implement the PostgreSQL event processor and make an important update to

`application.conf`.

22. PostgreSQL Event Processor

Let's begin by factoring out common event processor code and move the Cassandra-specific code to the new `com.chariotsolutions.tohlagom.cassandra` package. The following will be in `com.chariotsolutions.tohlagom.common`:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/common/HeroEventProcessor.scala

```
package com.chariotsolutions.tohlagom.common

import com.chariotsolutions.tohlagom.impl.HeroEvent
import com.lightbend.lagom.scaladsl.persistence.{AggregateEventShards,
AggregateEventTag}

object HeroEventProcessor {
  val Table = "hero"
  val IdColumn = "id"
  val NameColumn = "name"
  val EventProcessorId = "hero-offset"
}

trait HeroEventProcessor {
  def aggregateTags: Set[AggregateEventTag[HeroEvent]] = HeroEvent.Tag match {
    case tagger: AggregateEventTag[HeroEvent] =>
      Set(tagger)
    case shardedTagger: AggregateEventShards[HeroEvent] =>
      shardedTagger.allTags
  }
}
```

The Cassandra event processor should look recognizable by now:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/cassandra/HeroEventProcessor.scala

```
package com.chariotsolutions.tohlagom.cassandra

import akka.Done
import scala.concurrent.{ExecutionContext, Future, Promise}
import com.lightbend.lagom.scaladsl.persistence.ReadSideProcessor
import com.lightbend.lagom.scaladsl.persistence.cassandra.{CassandraReadSide,
CassandraSession}
import com.datastax.driver.core.PreparedStatement
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.impl._

class HeroEventProcessor(readSide: CassandraReadSide, session: CassandraSession
)(implicit ec: ExecutionContext)
  extends ReadSideProcessor[HeroEvent] with common.HeroEventProcessor {
  import common.HeroEventProcessor._
```

```
  // The promises are initialized in builder.setPrepare below.
  private val promiseInsert = Promise[PreparedStatement]
  private val promiseUpdate = Promise[PreparedStatement]
  private val promiseDelete = Promise[PreparedStatement]

  private def stmtInsert: Future[PreparedStatement] = promiseInsert.future
  private def stmtUpdate: Future[PreparedStatement] = promiseUpdate.future
  private def stmtDelete: Future[PreparedStatement] = promiseDelete.future

  def buildHandler(): ReadSideProcessor.ReadSideHandler[HeroEvent] = {
    val builder = readSide.builder[HeroEvent](EventProcessorId)

    builder.setGlobalPrepare(() =>
      for {
        _ <- session.executeCreateTable(
        s"""CREATE TABLE IF NOT EXISTS $Table (
           |  $IdColumn TEXT,
           |  $NameColumn TEXT,
           |  PRIMARY KEY ($IdColumn)
           |)""".stripMargin)
        _ <- session.executeWrite( s"""CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix |
ON $Table ($NameColumn) | USING 'org.apache.cassandra.index.sasi.SASIIndex'"""
.stripMargin) } yield Done ) builder.setPrepare { _ =>
      val insertStmt = session.prepare(s"INSERT INTO $Table ($IdColumn, $NameColumn)
VALUES (?, ?)")
      val updateStmt = session.prepare(s"UPDATE $Table SET $NameColumn = ? WHERE
$IdColumn = ?")
      val deleteStmt = session.prepare(s"DELETE FROM $Table WHERE $IdColumn = ?")

      promiseInsert.completeWith(insertStmt)
      promiseUpdate.completeWith(updateStmt)
      promiseDelete.completeWith(deleteStmt)

      for {
        _ <- insertStmt
        _ <- updateStmt
        _ <- deleteStmt } yield Done } builder.setEventHandler[HeroCreated] { element
=>
      stmtInsert.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.name)
        List(stmt)
      }
    }

    builder.setEventHandler[HeroChanged] { element =>
      stmtUpdate.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        stmt.setString(NameColumn, element.event.newName)
        List(stmt)
      }
```

```
    }

    builder.setEventHandler[HeroDeleted.type] { element =>
      stmtDelete.map(_.bind).map { stmt =>
        stmt.setString(IdColumn, element.entityId)
        List(stmt)
      }
    }

    builder.build
  }
}
```

The PostgreSQL event processor will differ from it in two main respects. First, indexing for prefix searching will be different. Now, we could just define an index for the `name` column. In fact, unlike Cassandra, a PostgreSQL database won't refuse to perform a search by a column even if no index for that column is defined; it will just be slow. However, a standard index, while improving an exact match search, will not be most efficient for prefix search. For that, a special kind of index—a trigram-based index—should be defined. This is how the global prepare callback would look like for a PostgreSQL event processor:

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/HeroEventProcessor.scala

```
    builder.setGlobalPrepare { connection =>
      JdbcSession.tryWith(connection.prepareStatement(
        s"""CREATE TABLE IF NOT EXISTS $Table (
           |  $IdColumn VARCHAR(10),
           |  $NameColumn VARCHAR(256),
           |  PRIMARY KEY ($IdColumn)
           |)""".stripMargin))(_.execute())
      JdbcSession.tryWith(connection.prepareStatement(
        "CREATE EXTENSION IF NOT EXISTS pg_trgm"))(_.execute())
      JdbcSession.tryWith(connection.prepareStatement(
        s"""CREATE INDEX IF NOT EXISTS fn_prefix
           |  ON $Table USING gin ($NameColumn gin_trgm_ops)
           |""".stripMargin))(_.execute())
    }
```

The second difference is that we cannot use the prepare callback to pre-make `PreparedStatement` instances, as was done in the Cassandra event processor, because JDBC `PreparedStatement` is not thread-safe. Thus, statements must be prepared in the event handlers, instead. In the end, the `HeroEventProcessor.scala` file in the `com.chariotsolutions.tohlagom.mixed` package would look as follows.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/HeroEventProcessor.scala

```
package com.chariotsolutions.tohlagom.mixed

import com.lightbend.lagom.scaladsl.persistence.jdbc.{JdbcReadSide, JdbcSession}
```

```
import com.lightbend.lagom.scaladsl.persistence.ReadSideProcessor
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.impl._

class HeroEventProcessor(readSide: JdbcReadSide) extends ReadSideProcessor[HeroEvent]
with common.HeroEventProcessor {
  import common.HeroEventProcessor._

  def buildHandler(): ReadSideProcessor.ReadSideHandler[HeroEvent] = {
    val builder = readSide.builder[HeroEvent](EventProcessorId)

    builder.setGlobalPrepare { connection =>
      JdbcSession.tryWith(connection.prepareStatement(
        s"""CREATE TABLE IF NOT EXISTS $Table (
           |  $IdColumn VARCHAR(10),
           |  $NameColumn VARCHAR(256),
           |  PRIMARY KEY ($IdColumn)
           |)""".stripMargin))(_.execute())
      JdbcSession.tryWith(connection.prepareStatement(
        "CREATE EXTENSION IF NOT EXISTS pg_trgm"))(_.execute())
      JdbcSession.tryWith(connection.prepareStatement(
        s"""CREATE INDEX IF NOT EXISTS fn_prefix
           |  ON $Table USING gin ($NameColumn gin_trgm_ops)
           |""".stripMargin))(_.execute())
    }

    builder.setEventHandler[HeroCreated] { case (connection, element) =>
      JdbcSession.tryWith(
        connection.prepareStatement(s"INSERT INTO $Table ($IdColumn, $NameColumn)
VALUES (?, ?)")
      ) { statement =>
        statement.setString(1, element.entityId)
        statement.setString(2, element.event.name)
        statement.execute()
      }
    }

    builder.setEventHandler[HeroChanged] { case (connection, element) =>
      JdbcSession.tryWith(
        connection.prepareStatement(s"UPDATE $Table SET $NameColumn = ? WHERE
$IdColumn = ?")
      ) { statement =>
        statement.setString(1, element.event.newName)
        statement.setString(2, element.entityId)
        statement.execute()
      }
    }

    builder.setEventHandler[HeroDeleted.type] { case (connection, element) =>
      JdbcSession.tryWith(
        connection.prepareStatement(s"DELETE FROM $Table WHERE $IdColumn = ?")
```

```
      ) { statement =>
        statement.setString(1, element.entityId)
        statement.execute()
      }
    }

    builder.build
  }
}
```

23. Running in Mixed Persistence Mode

Unless you already have a PostgreSQL database installed locally, you'll need to download and install one on your machine. We will need it when running our Lagom microservice in the development mode.

But first, we need to make a few changes to `application.conf`. To begin with, we must point it to the mixed-mode application loader. Then, we also need to make sure that Cassandra is used for the journal and for the snapshot store. Lagom's Cassandra and JDBC persistence modules both define these plugins. Therefore, we must specify them explicitly in `application.conf`. We only need to specify PostgreSQL configuration, which we will do in a separate `postgresql.conf` file.

After all these changes, `application.conf` will look as follows.

toh-lagom-impl/src/main/resources/application.conf

```
include "cassandra"
include "postgresql"

akka {
  actor.serialization-bindings {
    "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
  }

  persistence {
    journal.plugin = cassandra-journal
    snapshot-store.plugin = cassandra-snapshot-store
  }
}

play.application.loader = com.chariotsolutions.tohlagom.mixed.TourOfHeroesLoader
```

whereas `postgresql.conf` will be defined like this:

toh-lagom-impl/src/main/resources/postgresql.conf

```
jdbc-defaults.slick.profile = "slick.jdbc.PostgresProfile$"

db.default {
  driver = "org.postgresql.Driver"
  username = "postgres"
  username = ${?POSTGRESQL_USERNAME}
  password = ${POSTGRESQL_PASSWORD}
  url = "jdbc:postgresql://localhost/toh_lagom"
  url = ${?POSTGRESQL_URL}
}
```

Just as we have done for the Cassandra credentials earlier, make sure to hide your PostgreSQL user's credentials into environment variables. The question mark means that if the environment

variable isn't defined, the previous value (if any) is to be used. This way, we can provide `postgres` as the default username and `jdbc:postgresql://localhost/toh_lagom` as the default URL.

We will also modify `application.conf` in `toh-lagom-impl/src/it/resources`, so that the integration tests continue using Cassandra both for the write side and the read side:

toh-lagom-impl/it/main/resources/application.conf

```
cassandra.default.keyspace = toh_lagom_test
cassandra-journal.keyspace = ${cassandra.default.keyspace}
cassandra-snapshot-store.keyspace = ${cassandra.default.keyspace}
lagom.persistence.read-side.cassandra.keyspace = ${cassandra.default.keyspace}

akka.actor.serialization-bindings {
  "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
}

play.application.loader = com.chariotsolutions.tohlagom.cassandra.TourOfHeroesLoader
```

As you can see from the JDBC URL in `postgresql.conf`, we require a `toh_lagom` database to exist in our local PostgreSQL database cluster. So, please create one. Also, delete the `hero` and `offsetstore` tables in AWS Cassandra. Those were automatically created for the Cassandra read side and will no longer be needed.

Now, when you run the microservice in the development mode, everything should work—running against AWS Cassandra as the write side and the local PostgreSQL database as the read side. In fact, if you had any events saved in the AWS Cassandra journal, you should now see them applied to the read side—demonstrating that, indeed, if you blow your read view entirely, it will be automatically restored.

The time came to think of moving our Tour of Heroes microservice to production.

In order to work in production, a Lagom microservice must know how to join an Akka cluster, for scalability and redundancy. Different microservices can coexist in the same cluster, where they mutually discover each other. Remote Akka actors from different services and from the same service running on different nodes can communicate as if they were all resided on the same machine.

In the development mode, Lagom takes care of setting up a simple cluster with a single node and no service discovery. But we will have to do some lifting for the production mode. There are several ways to configure Akka clusters, but we will go with Akka Discovery with Kubernetes API. Kubernetes is a popular orchestration platform that automates deployment and scaling of applications running as Docker containers. As we will see, it is easy to build Lagom microservices as Docker images, and Lagom is well integrated with Kubernetes—so much so that Kubernetes became a platform of choice for deploying Lagom microservices, along with OpenShift and Mesosphere DC/OS. With Kubernetes making it easy to assemble Akka clusters, it fits Lagom like a glove.

Before we configure our `build.sbt` to build Docker images, however, we will first make sure our microservice knows how to join an Akka cluster in Kubernetes. Define the Kubernetes API

dependency in `build.sbt`:

build.sbt

```
val kubernetesApi = "com.lightbend.akka.discovery" %% "akka-discovery-kubernetes-api"
% "1.0.5"
```

Then, add it and the Akka Discovery library provided by Lagom to the list of dependencies for the `toh-lagom-impl` subproject:

build.sbt

```
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      lagomScaladslPersistenceJdbc,
      lagomScaladslAkkaDiscovery,
      kubernetesApi % Runtime,
      postgresql % Runtime,
      macwire % Provided
    )
  )
```

Create a `production.conf` file under `toh-lagom-impl/src/main/resources` with the following:

toh-lagom-impl/src/main/resources/production.conf

```
include "application"

akka {
  discovery.kubernetes-api.pod-label-selector = "app=%s"

  management.cluster.bootstrap.contact-point-discovery {
    discovery-method = kubernetes-api
    required-contact-point-nr = 2
    service-name = "toh-lagom"
  }
}
```

Here, `required-contact-point-nr` is the number of peer nodes (including self) that any given node must discover in order to join a cluster. We set it to the minimum allowed, which is two. It also happens to be the default, so we could omitted the setting. But it helps to know how to change it.

We will point to `production.conf` with the `-Dconfig.resource` Java option when running in the production mode.

Now is the time to implement the service locator in the `load` method of the `TourOfHeroLoader` class, which we do by mixing in Lagom's `AkkaDiscoveryComponents`.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/TourOfHeroesLoader.scala

```
  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with AkkaDiscoveryComponents
```

Here is how `TourOfHeroesLoader.scala` would look in the `com.chariotsolutions.tohlagom.mixed` package. But do this also in the `com.chariotsolutions.tohlagom.cassandra` package.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/mixed/TourOfHeroesLoader.scala

```
package com.chariotsolutions.tohlagom.mixed

import com.softwaremill.macwire.wire
import play.api.db.HikariCPComponents
import com.lightbend.lagom.scaladsl.devmode.LagomDevModeComponents
import com.lightbend.lagom.scaladsl.akka.discovery.AkkaDiscoveryComponents
import com.lightbend.lagom.scaladsl.server.{LagomApplication, LagomApplicationContext}
import com.lightbend.lagom.scaladsl.persistence.jdbc.ReadSideJdbcPersistenceComponents
import
com.lightbend.lagom.scaladsl.persistence.cassandra.WriteSideCassandraPersistenceCompon
ents
import com.chariotsolutions.tohlagom.common
import com.chariotsolutions.tohlagom.api._

class TourOfHeroesLoader extends common.TourOfHeroesLoader {
  def load(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with AkkaDiscoveryComponents

  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new TourOfHeroesApplication(context) with LagomDevModeComponents
}

abstract class TourOfHeroesApplication(context: LagomApplicationContext)
  extends LagomApplication(context) with common.TourOfHeroesApplication
    with ReadSideJdbcPersistenceComponents with HikariCPComponents
    with WriteSideCassandraPersistenceComponents {
  lazy val lagomServer = serverFor[TourOfHeroesService](wire[TourOfHeroesServiceImpl])
}
```

Part V. Production Mode

24. Production Configuration

Whereas Akka Discovery based on Kubernetes API, as configured above, works for the microservices to discover each other to form an Akka cluster, our Cassandra service won't run in the same Kubernetes cluster. The `lagomUnmanagedServices` setting in `build.sbt` only works in the development mode.

Lagom provides instructions on how to configure Cassandra static endpoints. This is done as follows:

```
cassandra.default {
  ## list the contact points  here
  contact-points = ["10.0.1.71", "23.51.143.11"]
  ## override Lagom□s ServiceLocator-based ConfigSessionProvider
  session-provider = akka.persistence.cassandra.ConfigSessionProvider
}

cassandra-journal {
  contact-points = ${cassandra.default.contact-points}
  session-provider = ${cassandra.default.session-provider}
}

cassandra-snapshot-store {
  contact-points = ${cassandra.default.contact-points}
  session-provider = ${cassandra.default.session-provider}
}

lagom.persistence.read-side.cassandra {
  contact-points = ${cassandra.default.contact-points}
  session-provider = ${cassandra.default.session-provider}
}
```

Unfortunately, `cassandra.us-east-1.amazonaws.com:9142` won't work as a static endpoint. If you kept the microservice running against AWS Cassandra in the development mode for a long time, you may have noticed that, after a while, the AWS Cassandra connection goes stale and the microservice doesn't reconnect. This is because `akka.persistence.cassandra.ConfigSessionProvider` resolves the contact point to an IP address, and AWS Cassandra recycles IP addresses over time.

So, we must develop a simple session provider of our own, instead. Create an `AmazonSessionProvider.scala` file in the `com.chariotsolutions.tohlagom.impl` package. The `AmazonSessionProvider` class extends Akka's `ConfigSessionProvider` and overrides its `lookupContactPoints` methods to use an unresolved `InetSocketAddress` as the contact point.

toh-lagom-impl/src/main/scala/com/chariotsolutions/tohlagom/impl/AmazonSessionProvider.scala

```
package com.chariotsolutions.tohlagom.impl

import akka.actor.ActorSystem
import java.net.InetSocketAddress
```

```
import scala.concurrent.{ExecutionContext, Future}
import akka.persistence.cassandra.ConfigSessionProvider
import com.typesafe.config.Config

class AmazonSessionProvider(system: ActorSystem, config: Config) extends
ConfigSessionProvider(system, config) {
  override def lookupContactPoints(clusterId: String)(implicit ec: ExecutionContext) =
{
    val region = config.getString("aws-region")
    val address = InetSocketAddress.createUnresolved(
s"cassandra.$region.amazonaws.com", 9142)
    Future.successful(List(address))
  }
}
```

All we need to do now is to set the session provider to our class and provide a value for the `aws-region` configuration setting in `production.conf`, as well as to change the Cassandra keyspace from `toh_lagom_dev` to `toh_lagom`, so as not to affect production when running in the development mode.

toh-lagom-impl/src/main/resources/production.conf

```
cassandra.default {
  keyspace = toh_lagom
  aws-region = "us-east-1"
  session-provider = com.chariotsolutions.tohlagom.impl.AmazonSessionProvider
}

cassandra-journal {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
}

cassandra-query-journal {
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
}

cassandra-snapshot-store {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
}

lagom.persistence.read-side.cassandra {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
}
```

Here, we have configured even the Cassandra read side, in anticipation of the time when we would be able to use either Cassandra-only or the mixed mode. Of course, we could have also gotten rid of the `aws-region` setting entirely, using the current AWS region instead. However, AWS Cassandra currently doesn't support replication across regions. So, it is conceivable—albeit, not terribly efficient—for AWS Kubernetes clusters in multiple regions to point to an AWS Cassandra in a different region. By making the `aws-region` setting configurable, we retain this flexibility (and still don't have to use AWS Java SDK as a dependency for the main project, besides).

Next, we will use AWS Secrets Manager to secure both the AWS Cassandra and the PostgreSQL database credentials.

25. Securing Database Credentials

It is a bad practice to commit your database credentials into a source code repository. This is why we have been using environment variables for the AWS Cassandra and PostgreSQL database credentials, referencing them in the configuration files:

- `AWS_MCS_USERNAME` and `AWS_MCS_PASSWORD` for AWS Cassandra;
- `POSTGRESQL_USERNAME` and `POSTGRESQL_PASSWORD` for PostgreSQL database.

However, as we move to production, we must find a good way to populate these settings automatically. One way to do this is by using AWS Secrets Manager. This involves a few manual steps to enter those credentials to the Secrets Manager. Following that, we will write a simple helper to retrieve them during the build time.

In the AWS console for Secrets Manager, create a new secret, as follows:

- Click the **Store a new secret** button.
- Select **Other type of secrets** for the secret type.
- Add a second row by clicking + **Add row**.
- For the first row, enter `username` as the key and the Cassandra username as the value.
- For the second row, enter `password` as the key and the Cassandra password as the value.
- Click **Next**, accepting the defaults.
- Enter `toh-lagom-cassandra` as the secret's name.
- Enter whatever description and tags that you wish, then click **Next**.
- We won't enable secret rotation. So, accept the defaults and click **Next**.
- Click the **Store** button, saving the AWS Cassandra credentials.

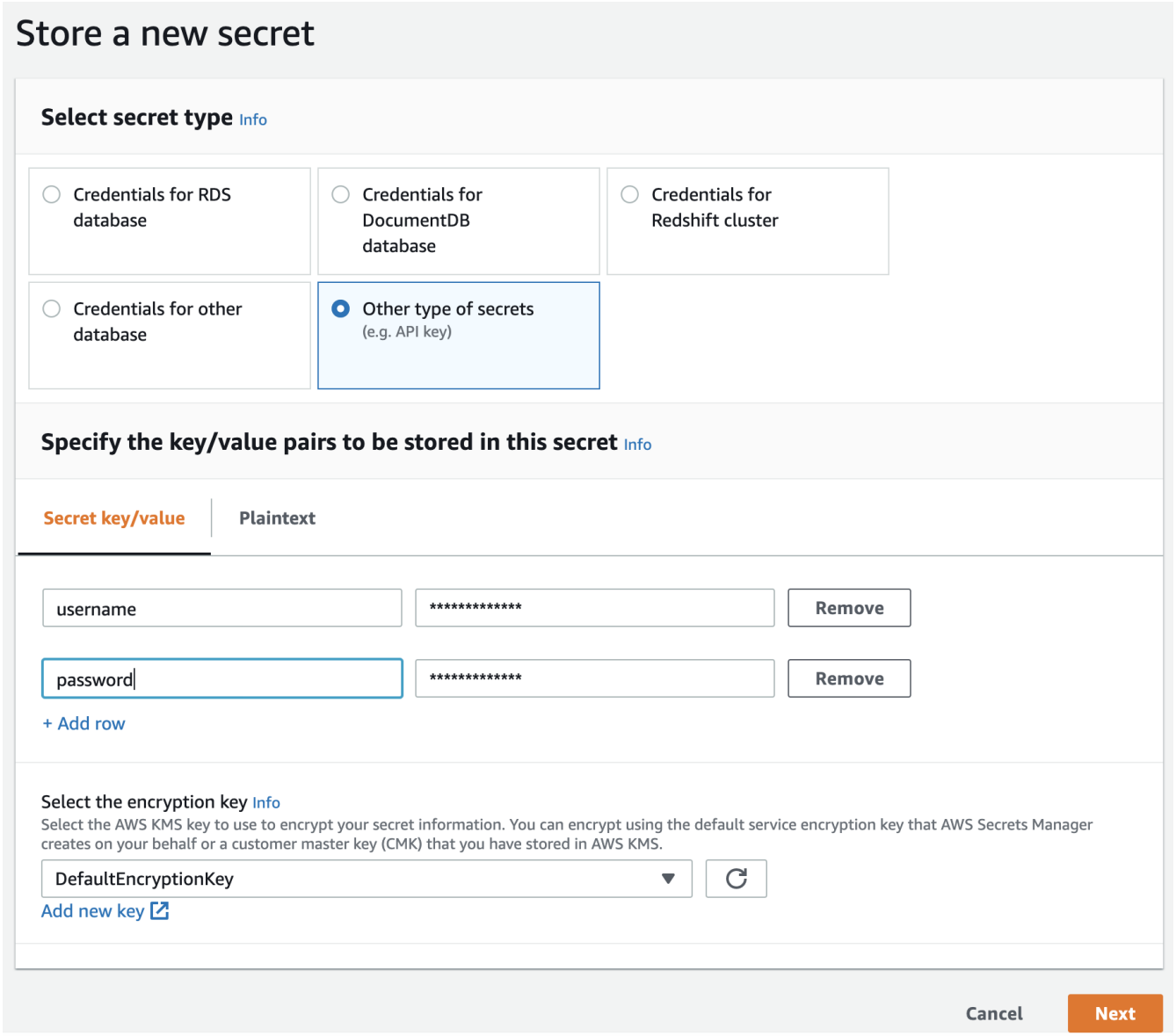


Figure 4. Storing a New Secret

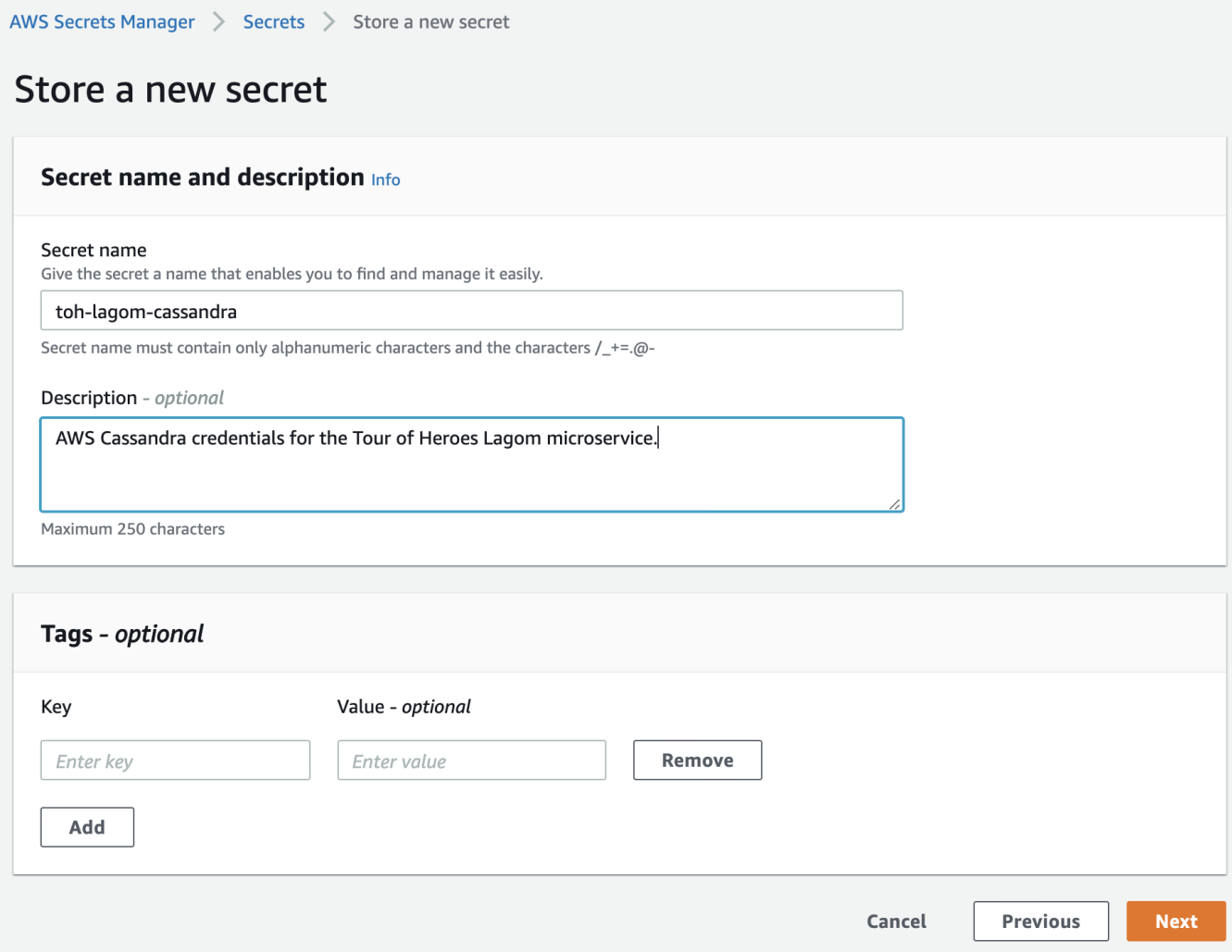


Figure 5. Storing a New Secret, continued

In the same way, create a `toh-lagom-postgresql` secret, with the PostgreSQL database credentials.

```
Here, we create a secret to be used with an Aurora database cluster, but with
the *Other type of secrets* type, rather than the *Credentials for RDS database*.
This is because we haven't created our database cluster yet.

A different way to handle this is to generate the credentials from the deployment
script (to be introduced later). That way, the secret would be created from the script
for an already existing database, which also makes it easier to enable secret
rotation.
This can be accomplished by defining an https://aws.amazon.com/lambda/[AWS Lambda] to
both update the Aurora database credentials _and_ the environment variables in the
Kubernetes pods. That is arguably a better way, but we will keep things simple and
not rotate the secrets.
```

The simple helper utility to retrieve these credentials will be called from `build.sbt`, so we will place it in the `project` folder. Since AWS Java SDK returns the secrets in the Json format, we will need a library to parse it. Add the Play Json library as a dependency to `project/build.sbt`:

project/build.sbt

```
libraryDependencies ++= Seq(
  "com.amazonaws" % "aws-java-sdk" % "1.11.762",
  "com.typesafe.play" %% "play-json" % "2.8.1"
)
```

Then, create a `project/AmazonUtils.scala` file, with the following code:

project/AmazonUtils.scala

```
import com.amazonaws.util.Base64
import com.amazonaws.regions.{Region, Regions}
import com.amazonaws.services.secretsmanager.model.GetSecretValueRequest
import com.amazonaws.services.secretsmanager.AWSSecretsManagerClientBuilder
import play.api.libs.json.{Format, Json}

object AmazonUtils {
  case class Credentials(username: String, password: String)
  implicit val format: Format[Credentials] = Json.format

  lazy val awsRegionName = sys.env("AWS_REGION")
  lazy val awsRegion = Region.getRegion(Regions.fromName(awsRegionName))
  private lazy val awsSecretsClient = AWSSecretsManagerClientBuilder.standard
    .withRegion(awsRegionName).build

  lazy val cassandraCredentials = getCredentials("toh-lagom-cassandra")
  lazy val postgresqlCredentials = getCredentials("toh-lagom-postgresql")

  private def getCredentials(secretId: String): Credentials = {
    val request = new GetSecretValueRequest().withSecretId(secretId)
    val result = awsSecretsClient.getSecretValue(request)
    val secret = Option(result.getSecretString)
      .getOrElse(new String(Base64.decode(result.getSecretBinary.array)))
    Json.parse(secret).as[Credentials]
  }
}
```

Here, we introduce a simple `Credentials` class to store username and password and define a Json format to deserialize it. Then, we determine the region and initialize an AWS Secrets client for that region. Then, we write a private `getCredentials` method to retrieve secrets of a given name by making AWS Java SDK calls and deserializing the returned Json as an instance of `Credentials`. Then, we provide handles to the two specific secrets we have created manually above.

We'll see shortly how these secrets are provided to the Docker container at runtime. But they aren't baked into the Docker images at build time.

It may seem redundant to save the AWS Cassandra service-specific credentials also into AWS Secrets Manager. But not only does this offer a generic way to handle all

credentials, both for AWS Cassandra and the PostgreSQL database, but it allows the added flexibility of another user running the deployment script than the one whose AWS Cassandra credentials are being used.

26. AWS Cassandra Schema

Until now, the Cassandra keyspace and tables were automatically created for us. This is convenient for working in the development mode, but even then we saw how a few initial requests could be lost while the `snapshots` table was being created (unlike the other tables, it is created upon the first request, not when the microservice is initialized). It is even worse in the production mode, however, since Cassandra is not well suited for concurrent schema updates. When multiple nodes running our microservice in the AWS Kubernetes cluster will begin creating the same tables, subtle problems can arise that may be difficult to troubleshoot.

Lagom recommends disabling auto-creating of the keyspace and the tables in production. This is done by setting the `keyspace-autocreate` and `tables-autocreate` settings in `production.conf` to `false`:

toh-lagom-impl/src/main/resources/production.conf

```
include "application"

akka {
  discovery.kubernetes-api.pod-label-selector = "app=%s"

  management.cluster.bootstrap.contact-point-discovery {
    discovery-method = kubernetes-api
    required-contact-point-nr = 2
    service-name = "toh-lagom"
  }
}

cassandra.default {
  keyspace = toh_lagom
  aws-region = "us-east-1"
  session-provider = com.chariotsolutions.tohlagom.impl.AmazonSessionProvider
}

cassandra-journal {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
  keyspace-autocreate = false
  tables-autocreate = false
}

cassandra-query-journal {
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
}

cassandra-snapshot-store {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
```

```
  keyspace-autocreate = false
  tables-autocreate = false
}

lagom.persistence.read-side.cassandra {
  keyspace = ${cassandra.default.keyspace}
  aws-region = ${cassandra.default.aws-region}
  session-provider = ${cassandra.default.session-provider}
  keyspace-autocreate = false
}
```

Instead, we will create the AWS Cassandra schema manually. This can be done in many ways; for example, by writing a Python script (DataStax Cassandra Python driver is easy to use and popular). But we will use this as a teaching moment to show how to add a custom SBT task. First, add the DataStax Java driver for Cassandra to `project/build.sbt` as a dependency:

project/build.sbt

```
libraryDependencies ++= Seq(
  "com.amazonaws" % "aws-java-sdk" % "1.11.762",
  "com.typesafe.play" %% "play-json" % "2.8.1",
  "com.datastax.cassandra" % "cassandra-driver-core" % "3.8.0"
)
```

Then, create a `project/CassandraUtils.scala` file, where we will create the schema. AWS Cassandra currently supports only one replication strategy, `SingleRegionStrategy`, so we will use that one. Any other one will simply be ignored. The replication factor is also always set to 3, so we won't specify it when creating the keyspace.

project/CassandraUtils.scala

```
import scala.util.Try
import java.net.InetSocketAddress
import com.datastax.driver.core.{Cluster, PlainTextAuthProvider}

object CassandraUtils {
  lazy val contactPoint = s"""cassandra.${sys.env("AWS_REGION")}.amazonaws.com"""
  val keyspace = "toh_lagom"
  val port = 9142

  def initializeSchema(): Unit = {
    System.setProperty("javax.net.ssl.trustStore", "../cassandra_truststore.jsk")
    System.setProperty("javax.net.ssl.trustStorePassword", "amazon")
    val credentials = AmazonUtils.cassandraCredentials

    val authProvider = new PlainTextAuthProvider(credentials.username, credentials
.password)
    val address = new InetSocketAddress(contactPoint, port)
    val cluster = Cluster.builder.addContactPoints(address.getAddress)
      .withPort(port).withAuthProvider(authProvider).withSSL.build
```

```
    val session = cluster.newSession

    Try {
      session.execute(
        s"""CREATE KEYSPACE IF NOT EXISTS $keyspace
           |  WITH replication = {
           |    'class': 'SingleRegionStrategy'
           |  };
           |""".stripMargin)

      Thread.sleep(10000)
      println(s"Successfully created Cassandra keyspace $keyspace.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.messages (
           |  persistence_id text,
           |  partition_nr bigint,
           |  sequence_nr bigint,
           |  timestamp timeuuid,
           |  timebucket text,
           |  writer_uuid text,
           |  ser_id int,
           |  ser_manifest text,
           |  event_manifest text,
           |  event blob,
           |  message blob,
           |  meta_ser_id int,
           |  meta_ser_manifest text,
           |  meta blob,
           |  used boolean,
           |  tags set<text>,
           |  PRIMARY KEY ((persistence_id, partition_nr), sequence_nr, timestamp,
timebucket)
           |) WITH gc_grace_seconds = 864000
           |  AND compaction = {
           |    'class': 'SizeTieredCompactionStrategy',
           |    'enabled': true,
           |    'tombstone_compaction_interval': 86400,
           |    'tombstone_threshold': 0.2,
           |    'unchecked_tombstone_compaction': false,
           |    'bucket_high': 1.5,
           |    'bucket_low': 0.5,
           |    'max_threshold': 32,
           |    'min_threshold': 4,
           |    'min_sstable_size': 50
           |  };
           |""".stripMargin)
      println(s"Successfully created Cassandra table $keyspace.messages.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.tag_views (
```

```
            |  tag_name text,
            |  persistence_id text,
            |  sequence_nr bigint,
            |  timebucket bigint,
            |  timestamp timeuuid,
            |  tag_pid_sequence_nr bigint,
            |  writer_uuid text,
            |  ser_id int,
            |  ser_manifest text,
            |  event_manifest text,
            |  event blob,
            |  meta_ser_id int,
            |  meta_ser_manifest text,
            |  meta blob,
            |  PRIMARY KEY ((tag_name, timebucket), timestamp, persistence_id,
tag_pid_sequence_nr)
            |) WITH gc_grace_seconds = 864000
            |  AND compaction = {
            |    'class': 'SizeTieredCompactionStrategy',
            |    'enabled': true,
            |    'tombstone_compaction_interval': 86400,
            |    'tombstone_threshold': 0.2,
            |    'unchecked_tombstone_compaction': false,
            |    'bucket_high': 1.5,
            |    'bucket_low': 0.5,
            |    'max_threshold': 32,
            |    'min_threshold': 4,
            |    'min_sstable_size': 50
            |  };
            |""".stripMargin)
      println(s"Successfully created Cassandra table $keyspace.tag_views.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.tag_write_progress(
            |  persistence_id text,
            |  tag text,
            |  sequence_nr bigint,
            |  tag_pid_sequence_nr bigint,
            |  offset timeuuid,
            |  PRIMARY KEY (persistence_id, tag)
            |);
            |""".stripMargin)
      println(s"Successfully created Cassandra table $keyspace.tag_write_progress.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.tag_scanning(
            |  persistence_id text,
            |  sequence_nr bigint,
            |  PRIMARY KEY (persistence_id)
            |);
            |""".stripMargin)
```

```
      println(s"Successfully created Cassandra table $keyspace.tag_scanning.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.metadata(
           |  persistence_id text PRIMARY KEY,
           |  deleted_to bigint,
           |  properties map<text,text>
           |);
           |""".stripMargin)
      println(s"Successfully created Cassandra table $keyspace.metadata.")

      session.execute(
        s"""CREATE TABLE IF NOT EXISTS $keyspace.snapshots (
           |  persistence_id text,
           |  sequence_nr bigint,
           |  timestamp bigint,
           |  ser_id int,
           |  ser_manifest text,
           |  snapshot_data blob,
           |  snapshot blob,
           |  meta_ser_id int,
           |  meta_ser_manifest text,
           |  meta blob,
           |  PRIMARY KEY (persistence_id, sequence_nr)
           |) WITH CLUSTERING ORDER BY (sequence_nr DESC) AND gc_grace_seconds =
864000
           |  AND compaction = {
           |    'class': 'SizeTieredCompactionStrategy',
           |    'enabled': true,
           |    'tombstone_compaction_interval': 86400,
           |    'tombstone_threshold': 0.2,
           |    'unchecked_tombstone_compaction': false,
           |    'bucket_high': 1.5,
           |    'bucket_low': 0.5,
           |    'max_threshold': 32,
           |    'min_threshold': 4,
           |    'min_sstable_size': 50
           |  };
           |""".stripMargin)
      println(s"Successfully created Cassandra table $keyspace.snapshots.")
    } recover { case error =>
      error.printStackTrace()
      session.close()
      cluster.close()
      throw error
    }

    session.close()
    cluster.close()
  }
```

```
}
```

The schema definitions are the default ones taken from the Akka Persistence documentation here and here, except that the `messages` table has extra `message` and `used` columns not reflected in the documentation. After creating a keyspace, we include a wait in order to make sure that AWS propagates the changes properly. Also, now that we have factored out the AWS Cassandra endpoint into the `CassandraUtils.contactPoint` method, we can now call it from `build.sbt` in all places where we previously used the string.

Now we can define a new SBT task in `build.sbt`:

build.sbt

```
lazy val initializeSchema = taskKey[Unit]("Initializes Cassandra schema")
initializeSchema := CassandraUtils.initializeSchema()
```

Running `sbt initializeSchema` now creates the AWS Cassandra keyspace and tables, if those have not been created yet. We'll invoke this task from our deployment script. But first, let's see how to build our microservice as a Docker image.

27. Building Docker Images

Begin by installing Docker if you don't have it already. Instructions for your operating system can be found here.

Lagom uses the SBT plugin called `sbt-native-packager` for production builds. Since the Lagom plugin added to `project/plugins.sbt` already depends on `sbt-native-packager`, we do not need to add it explicitly. The documentation for `sbt-native-packager` defined a number of different SBT tasks to package an application for production:

- `sbt universal:packageBin` to produce a ZIP distribution;
- `sbt rpm:packageBin` to produce an RPM package;
- `sbt docker:publishLocal` to add a Docker image to a local repository;
- `sbt docker:publish` to add a Docker image to a remote repository;
- and so on.

To configure a Docker build for our microservice, add the following call to the definition of the `toh-lagom-impl` subproject in `build.sbt`:

build.sbt

```
  .settings(
    dockerBaseImage := "adoptopenjdk/openjdk11",
    packageName in Docker := (name in ThisBuild).value,
    dockerExposedPorts in Docker := Seq(9000, 9008, 8558, 2552, 25520),
    mappings in Universal += file("cassandra_truststore.jks") ->
"cassandra_truststore.jks",
    javaOptions in Universal ++= Seq(
      "-Dpidfile.path=/dev/null",
      "-Dconfig.resource=production.conf",
      s"-Dplay.http.secret.key=${Random.alphanumeric.take(40).mkString}"
    )
  )
```

Here, we make sure that the Docker image will use Java 11 (the default is Java 8, but it has reached the end of life). If you decided to use Java 8, though, then don't include that line. Then, we set `packageName` to "toh-lagom" and make the Docker containers running from this image to expose a few important ports:

- 9000: the service gateway port
- 9008: the service locator port
- 8558: the Akka Management port (although we do not use it)
- 2552 and 25520: ports for communication between actors

Then, we add the trust store file, `cassandra_truststore.jks` to the Docker image by mapping it into the top-level folder in the Docker build hierarchy: `/opt/docker`. This way, our `cassandra.conf` settings

will work without change, since the trust store file is also at the top of the project hierarchy.

Then, we set our Java options, pointing to `production.conf` as `config.resource`. This allows us to continue using `application.conf` in the development mode. The `pidfile.path` setting is necessary to prevent the following error when running the microservice in the production mode:

```
java.nio.file.AccessDeniedException: /opt/docker/RUNNING_PID
```

The application secret setting, `play.http.secret.key`, is required in the production mode. In the development mode, it is set to "changeme"; but if it isn't set or is set to that default value in the production mode, the application will refuse to run. Anyone with access to the application secret will be able to generate any session they please, effectively allowing them to log in to your system as any user. So, the application secret must be secure. Here, we set it to a random alphanumeric string of length 40, without committing its value to source code.

When we now run `sbt docker:publishLocal`, a Docker image is built and published in the local Docker repository. The file system of a would-be-running Docker container can be examined under `toh-lagom-impl/target/docker/stage`.

28. Publishing to ECR

We haven't configured a remote Docker repository to publish to, because we will use Amazon ECR (Elastic Container Repository) for that purpose. Setting it up is rather more involved, but thankfully someone has already done the heavy lifting, creating the `sbt-ecr` plugin. So, we begin by adding it to our `project/plugins.sbt`:

project/plugins.sbt

```
addSbtPlugin("com.lightbend.lagom" % "lagom-sbt-plugin" % "1.6.1")
addSbtPlugin("com.mintbeans" % "sbt-ecr" % "0.15.0")
```

That done, we can now modify the `.enablePlugins` call on the `toh-lagom-impl` subproject in `build.sbt` to include that plugin together with Lagom:

build.sbt

```
  .enablePlugins(LagomScala, EcrPlugin)
```

Then, we add another `.setting` call to the `toh-lagom-impl` subproject, with the ECR-related settings:

build.sbt

```
  .settings(
    region in Ecr := AmazonUtils.awsRegion,
    repositoryTags in Ecr ++= Seq(version.value),
    repositoryName in Ecr := (packageName in Docker).value,
    login in Ecr := ((login in Ecr) dependsOn (createRepository in Ecr)).value,
    push in Ecr := ((push in Ecr) dependsOn (publishLocal in Docker, login in Ecr
)).value,
    localDockerImage in Ecr := s"${(packageName in Docker).value}:${(version in
Docker).value}"
  )
```

Here, we set the name and the AWS region for the ECR repository. The `latest` tag is automatically added to the latest image, but we also want to tag each image with the version, for ease of upgrading (or downgrading) to a specific version.

The `sbt ecr:createRepository` task automatically creates a repository in ECR, based on the name, region, and the AWS account. If a matching repository already exists, this task makes no change. The `sbt ecr:login` task invokes also `sbt ecr:createRepository`, and the `sbt ecr:login` task invokes both `sbt docker:publishLocal` and `sbt ecr:login`. So, a complete clean build that culminates with pushing the produced Docker image into ECR can be accomplished by the following command:

```
sbt clean ecr:push
```

Now that we have Docker images of our microservice in Amazon ECR, we can deploy them to AWS

Kubernetes.

Part VI. Deploying to AWS

29. Deployment Script

Creating a cluster in AWS Kubernetes manually is a very complex, involved process, where it is very easy to make subtle mistakes. Thankfully, someone has done the heavy lifting for us again. Amazon recommends using the `eksctl` command line utility to automate cluster creation. To begin with, following the instructions here to install `eksctl`. The usual Kubernetes utility, `kubectl`, will be installed together with `eksctl`, if you don't have it already.

Make sure that your version of `eksctl` is 0.17 or later, if you want to use managed node groups in AWS Kubernetes.

Because we will rely on `eksctl`, it makes sense to continue using the command line to automate the rest of deployment. This means using AWS CLI, of course, but also the `yq` command line utility, recommended by AWS for parsing the Json output returned by AWS CLI. We will also use `envsubst` to substitute environment variables in Kubernetes configuration files.

```
`envsubst` should already be available on Linux but may need to be installed on Mac OS
as pat of `gettext`:

brew install gettext
brew link --force gettext
```

You can find instructions for installing and using `jq` here. It is recommended by Amazon for the scenarios where the AWS CLI's own `--query` parameter does not suffice. We will have a couple of cases like that here, so we will use `jq` throughout, for consistency.

For the deployment-related files, created a `deployment` folder under the top-level `toh-lagom`. Our goal is to create an `awsdeploy.sh` script that takes the following command line arguments:

```
. awsdeploy.sh <version> [<s3-bucket-name>]
```

The version of the microservice is required. It corresponds to the tag with which the corresponding Docker image is tagged in the ECR repository. The other argument is optional. It is the name of the S3 bucket in which the static website assets, obtained by compiling the Angular tutorial, are found. If not provided, a new S3 bucket will be created.

If a Docker image for that version is found in the ECR repository, it will be deployed. If not, we will build it and push it to the repository first. The end result will be a CloudFront distribution's URL through which to access the deployed website (we won't use AWS Route 53 to expose our microservice through a registered domain, leaving this step to you if you wish to go the whole way). We will also produce an `awsupgrade.sh` script, also taking the version as a single argument. That script will upgrade the microservice in place and restart the pods in the Kubernetes cluster.

So, let's start writing the `awsdeploy.sh` script by saving the version into an environment variable. Then, we retrieve the AWS Cassandra and Aurora database credentials, to be used later in the script. As you will see, nowhere do we write them down into a file. Then, we create the ECR

repository, unless exists, and initialize the AWS Cassandra schema, if not yet done. The ECR repository would have been automatically created anyway during a build. But we must first look into the repository to see if the matching Docker image already exists and so whether a build is even necessary.

deployment/awsdeploy.sh

```
#!/usr/bin/env bash

export TOH_VERSION=$1
if [[ -z ${TOH_VERSION} ]]
then
    echo 'Must provide a version.'
    exit 1
else
    echo "Using the toh-lagom version ${TOH_VERSION}."
fi

echo "Retrieving AWS Cassandra credentials..."
cassandraSecret=$(aws secretsmanager get-secret-value --secret-id toh-lagom-cassandra
\
    | jq -r '.SecretString')
export AWS_MCS_USERNAME=$(echo ${cassandraSecret} | jq -r '.username')
export AWS_MCS_PASSWORD=$(echo ${cassandraSecret} | jq -r '.password')

echo "Retrieving Aurora database credentials..."
dbSecret=$(aws secretsmanager get-secret-value --secret-id toh-lagom-postgresql \
    | jq -r '.SecretString')
export POSTGRESQL_USERNAME=$(echo ${dbSecret} | jq -r '.username')
export POSTGRESQL_PASSWORD=$(echo ${dbSecret} | jq -r '.password')

echo "Initializing AWS Cassandra schema and ECR repository..."
cd ..; sbt ecr:createRepository initializeSchema; cd ./deployment
export REPOSITORY_URI=$(aws ecr describe-repositories --repository-names toh-lagom \
    | jq -r '.repositories[0].repositoryUri')
echo "AWS ECR repository: ${REPOSITORY_URI}"
```

Here, we can see how the `jq` command line utility works. For example, `jq` extracts the `repositoryUri` field from the first entry in the `repositories` array in the Json output returned by the `aws ecr describe-repositories` command. Now is the time to make the last change to our `build.sbt` to parameterize `version in ThisBuild` with the `TOH_VERSION` environment variable, defaulting to `1.0.0`.

build.sbt

```
import scala.util.Random

name in ThisBuild := "toh-lagom"
scalaVersion in ThisBuild := "2.13.1"
organization in ThisBuild := "com.chariotsolutions"
maintainer in ThisBuild := "lkorogodski@chariotsolutions.com"
```

```
version in ThisBuild := sys.env.getOrElse("TOH_VERSION", "1.0.0")

scalacOptions in ThisBuild ++= Seq(
  "-language:implicitConversions",
  "-language:postfixOps"
)

lagomKafkaEnabled in ThisBuild := false
lagomCassandraEnabled in ThisBuild := false
lagomUnmanagedServices in ThisBuild := Map(
  "cas_native" -> s"https://${CassandraUtils.contactPoint}:${CassandraUtils.port}"
)

val kubernetesApi = "com.lightbend.akka.discovery" %% "akka-discovery-kubernetes-api"
% "1.0.5"
val lagomTestKit = "com.lightbend.lagom" %% "lagom-scaladsl-testkit" % "1.6.1"
val macwire = "com.softwaremill.macwire" %% "macros" % "2.3.3"
val postgresql = "org.postgresql" % "postgresql" % "42.2.10"
val scalaTest = "org.scalatest" %% "scalatest" % "3.1.1"

lazy val `toh-lagom` = (project in file("."))
  .aggregate(`toh-lagom-api`, `toh-lagom-impl`)

lazy val `toh-lagom-api` = (project in file("toh-lagom-api"))
  .settings(libraryDependencies ++= Seq(lagomScaladslApi))

lazy val `toh-lagom-impl` = (project in file("toh-lagom-impl"))
  .enablePlugins(LagomScala, EcrPlugin)
  .dependsOn(`toh-lagom-api`)
  .configs(IntegrationTest)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      lagomScaladslPersistenceJdbc,
      lagomScaladslAkkaDiscovery,
      kubernetesApi % Runtime,
      postgresql % Runtime,
      macwire % Provided
    )
  )
  .settings(
    Defaults.itSettings,
    TestSettings.forked(IntegrationTest),
    libraryDependencies ++= Seq(
      scalaTest % Test,
      scalaTest % IntegrationTest,
      lagomTestKit % IntegrationTest
    )
  )
  .settings(
    dockerBaseImage := "adoptopenjdk/openjdk11",
```

```
    packageName in Docker := (name in ThisBuild).value,
    dockerExposedPorts in Docker := Seq(9000, 9008, 8558, 2552, 25520),
    mappings in Universal += file("cassandra_truststore.jks") ->
"cassandra_truststore.jks",
    javaOptions in Universal ++= Seq(
      "-Dpidfile.path=/dev/null",
      "-Dconfig.resource=production.conf",
      s"-Dplay.http.secret.key=${Random.alphanumeric.take(40).mkString}"
    )
  )
  .settings(
    region in Ecr := AmazonUtils.awsRegion,
    repositoryTags in Ecr ++= Seq(version.value),
    repositoryName in Ecr := (packageName in Docker).value,
    login in Ecr := ((login in Ecr) dependsOn (createRepository in Ecr)).value,
    push in Ecr := ((push in Ecr) dependsOn (publishLocal in Docker, login in Ecr
)).value,
    localDockerImage in Ecr := s"${(packageName in Docker).value}:${(version in
Docker).value}"
  )

lazy val initializeSchema = taskKey[Unit]("Initializes Cassandra schema")
initializeSchema := CassandraUtils.initializeSchema()
```

With this change, we can now add the Docker image build step to our deployment script.

deployment/awsdeploy.sh

```
if [[ -z $(aws ecr describe-images --repository-name toh-lagom --image-ids imageTag=
${TOH_VERSION}) ]]
then
    echo "The Docker image with tag ${TOH_VERSION} is NOT found."
    echo 'Building the ECR Docker image...'
    cd ..; sbt clean ecr:push; cd ./deployment
    echo 'Pushed the Docker image into AWS ECR.'
else
    echo "The Docker image with tag ${TOH_VERSION} is found."
fi
```

First, we check if a Docker image with the `TOH_VERSION` tag already exists in the ECR repository. If not, we build it. We don't have to wait for it to become available, however, because it will take far less time than creating an AWS Kubernetes cluster and an Aurora database cluster would, which is what we are about to do.

30. AWS Kubernetes Configuration

We will use the following YAML definition for our AWS Kubernetes cluster, which we will save in `deployment/k8cluster.yaml`:

deployment/k8cluster.yaml

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: toh-lagom
  region: ${AWS_REGION}
managedNodeGroups:
  - name: workers
    instanceType: t3.medium
    desiredCapacity: 2
    minSize: 2
    maxSize: 3
    labels:
      app: toh-lagom
      actorSystemName: toh-lagom
    ssh:
      allow: true
      #publicKeyPath: ... # will use ~/.ssh/id_rsa.pub as the default ssh key
availabilityZones:
  - ${AWS_REGION}a
  - ${AWS_REGION}f
```

For our simple deployment, we will use the minimal supported instance type, `t3.medium`. We will also use the minimal capacity of 2 and the maximal capacity of 3, just for illustration purposes. The `AWS_REGION` environment variable will be substituted with its value in the deployment script.

An AWS Kubernetes cluster requires at least two availability zones. If you don't specify any, they will be picked at random. Above is an example of how you can provide specific ones; for example, if you have insufficient capacity in some availability zones in your account.

As of the version 0.14, AWS Kubernetes supports managed node groups. The `eksctl` command line utility fully supports them starting at the version 0.17. If your version is earlier, replace `managedNodeGroups` above with `nodeGroups`.

We also must define an AWS Kubernetes deployment configuration, which we will do in the `deployment/k8deployment.yaml` file, as follows:

deployment/k8deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
```

```
    app: toh-lagom
  name: toh-lagom
spec:
  replicas: 2
  selector:
    matchLabels:
      app: toh-lagom
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: toh-lagom
        actorSystemName: toh-lagom
    spec:
      containers:
        - name: toh-lagom
          image: ${REPOSITORY_URI}:${TOH_TAG}
          imagePullPolicy: Always
          livenessProbe:
            httpGet:
              path: /alive
              port: management
          readinessProbe:
            httpGet:
              path: /ready
              port: management
         ports:
            - name: service-gateway
              containerPort: 9000
              protocol: TCP
            - name: service-locator
              containerPort: 9008
              protocol: TCP
            - name: remoting
              containerPort: 2552
              protocol: TCP
            - name: akka-artery
              containerPort: 25520
              protocol: TCP
            - name: management
              containerPort: 8558
              protocol: TCP
          env:
            # Kubernetes API discovery will use this service name
            # to look for nodes with this value in the 'app' label.
            - name: AKKA_CLUSTER_BOOTSTRAP_SERVICE_NAME
              valueFrom:
```

```
            fieldRef:
              apiVersion: v1
              fieldPath: "metadata.labels['app']"
        - name: POSTGRESQL_URL
          value: ${POSTGRESQL_URL}
        - name: POSTGRESQL_USERNAME
          value: ${POSTGRESQL_USERNAME}
        - name: POSTGRESQL_PASSWORD
          value: ${POSTGRESQL_PASSWORD}
        - name: AWS_MCS_USERNAME
          value: ${AWS_MCS_USERNAME}
        - name: AWS_MCS_PASSWORD
          value: ${AWS_MCS_PASSWORD}
```

Here, among other things, we declare the ports to be exposed and define the environment variables for the AWS Cassandra and Aurora credentials, as well as the Aurora database cluster URL, `POSTGRESQL_URL`. This way, we can pass the credentials safely, without writing them down into any file.

Kubernetes secrets is another way to pass credentials to the Kubernetes pods. For example, we could have defined the `AWS_MCS_USERNAME` and `AWS_MCS_PASSWORD` environment variables for the pods by reference to a Kubernetes secret, as follows:

```
        - name: AWS_MCS_USERNAME
          valueFrom:
            secretKeyRef:
              name: cassandra-secret
              key: password
        - name: AWS_MCS_PASSWORD
          valueFrom:
            secretKeyRef:
              name: cassandra-secret
              key: password
```

That would call for defining a `cassandra-secret` Kubernetes secret, though.

```
apiVersion: v1
kind: Secret
metadata:
  name: cassandra-secret
type: Opaque
data:
  username: ${AWS_MCS_USERNAME}
  password: ${AWS_MCS_PASSWORD}
```

This way, the username and password would have still been defined by substituting the values of environment variables in the deployment script. So, there is no advantage of doing so in our case, compared to simply doing this:

```
            - name: AWS_MCS_USERNAME
              value: ${AWS_MCS_USERNAME}
            - name: AWS_MCS_PASSWORD
              value: ${AWS_MCS_PASSWORD}
```

Kubernetes secrets can be useful, however, in more complex scenarios, so it is good to be aware of their availability. For example, if we wanted to roll over the secrets stored in AWS Secrets Manager while the Kubernetes pods are running, we could create an AWS Lambda that calls a Kubernetes Python client to update the Kubernetes secrets upon such rollover.

We also must define several roles and service accounts, without which our cluster won't work properly. These we will place in `deployment/k8roles.yaml`:

deployment/k8roles.yaml

```
# Create a role that can list pods and
# bind the default service account in the namespace
# that the binding is deployed to to that role.
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pod-reader
rules:
  - apiGroups: [""] # indicates the core API group
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
subjects:
  # Uses the default service account.
  # Consider creating a dedicated service account to run your
  # Akka Cluster services and binding the role to that one.
  - kind: ServiceAccount
    name: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: eks-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: eks-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: eks-admin
    namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fabric8-rbac
subjects:
  - kind: ServiceAccount
    name: default
    namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

31. AWS Kubernetes Cluster

Now we can create an AWS Kubernetes cluster by using the `eksctl` command line utility. But first, we substitute the environment variables' values in `k8cluster.yaml` with their values by using the `envsubst` utility. Let's add the following to our growing `awsdeploy.sh` script:

deployment/awsdeploy.sh

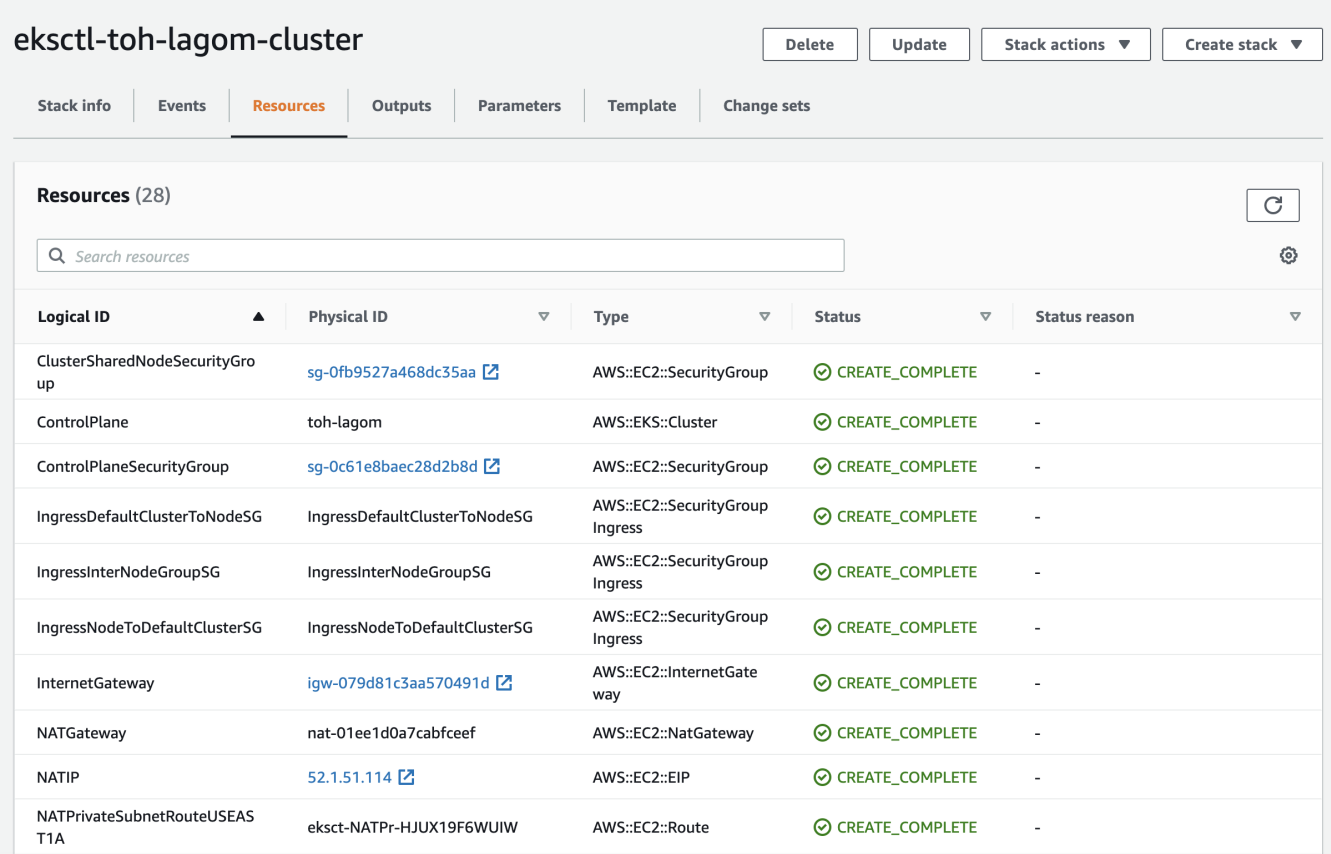
```
echo 'Creating AWS Kubernetes cluster...'
envsubst < k8cluster.yaml | eksctl create cluster -f -
kubectl apply -f k8roles.yaml

# To enable CloudWatch logging, execute this command:
#eksctl utils update-cluster-logging --region=${AWS_REGION} --cluster=toh-lagom

# For a GPU instance type and the Amazon EKS-optimized AMI with GPU support:
#echo 'Applying NVIDIA device plugin for Kubernetes...'
#kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/1.0.0-
beta/nvidia-device-plugin.yml
echo 'Successfully created an AWS Kubernetes cluster.'
```

Here, we also create the roles with `kubectl apply`. If you wish to enable CloudWatch logging for the cluster, uncomment the corresponding line above. If we had used an EKS-optimized AMI with GPU support, we would have also executed the commented out code to apply NVIDIA device plugin for Kubernetes.

We haven't loaded `k8deployment.yaml` yet, because we haven't obtained the value of our `POSTGRESQL_URL` environment variable yet. We can do it only after having created an Aurora database cluster.

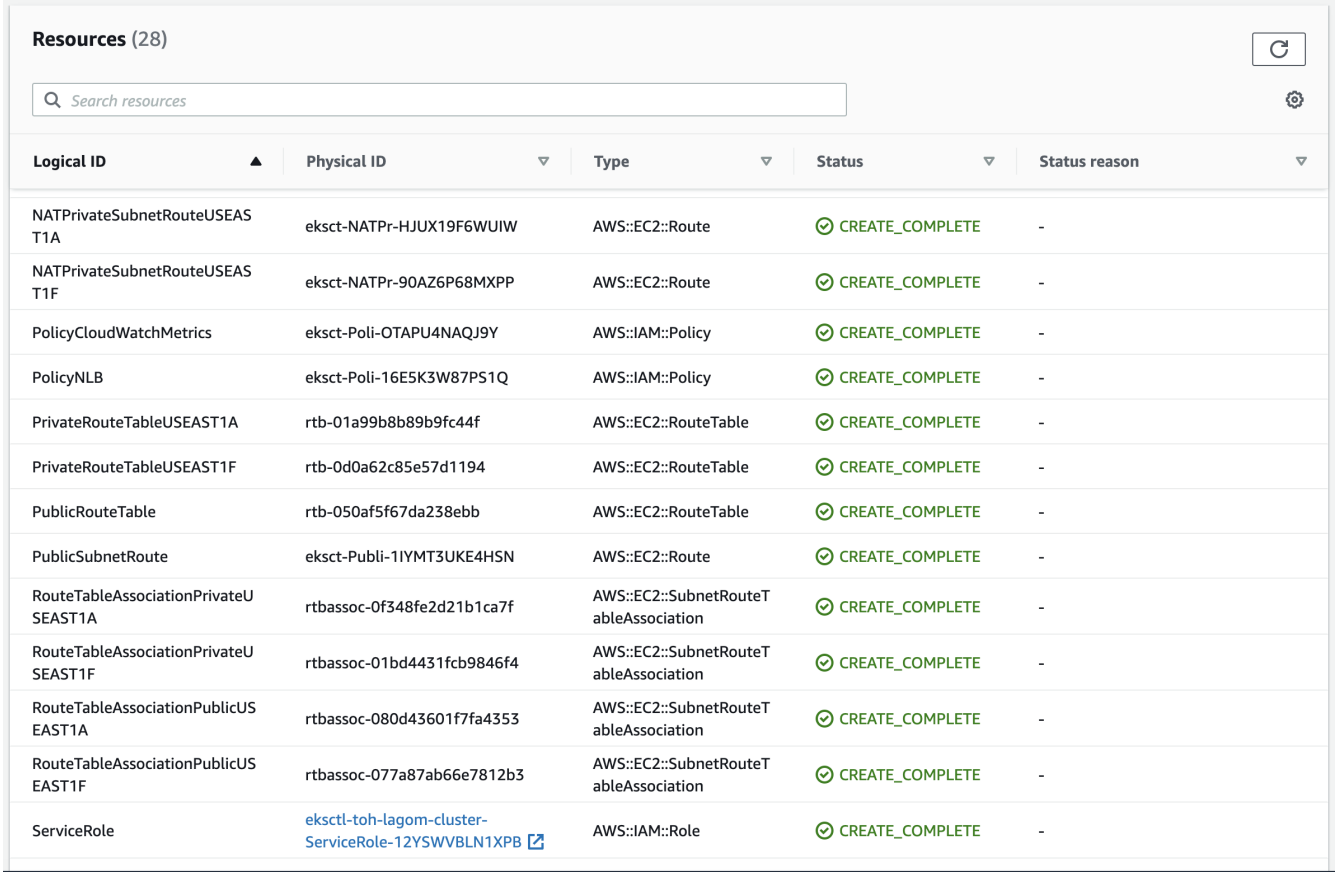
Optionally, for convenient monitoring of an AWS Kubernetes cluster, you can install a Kubernetes Dashboard with a web UI by following these instructions, and the Kubernetes Metrics Server by following these.

The `eksctl create cluster` command will take some considerable time to complete. When it is done, go to the AWS CloudFormation console to examine two stacks that we created in the background as a result: `eksctl-toh-lagom-cluster` and `eksctl-toh-lagom-nodegroup-workers`. It can be informative—and rather daunting—to imagine what we would have had to do if we went the manual way, by looking at the Resources tab of the former.



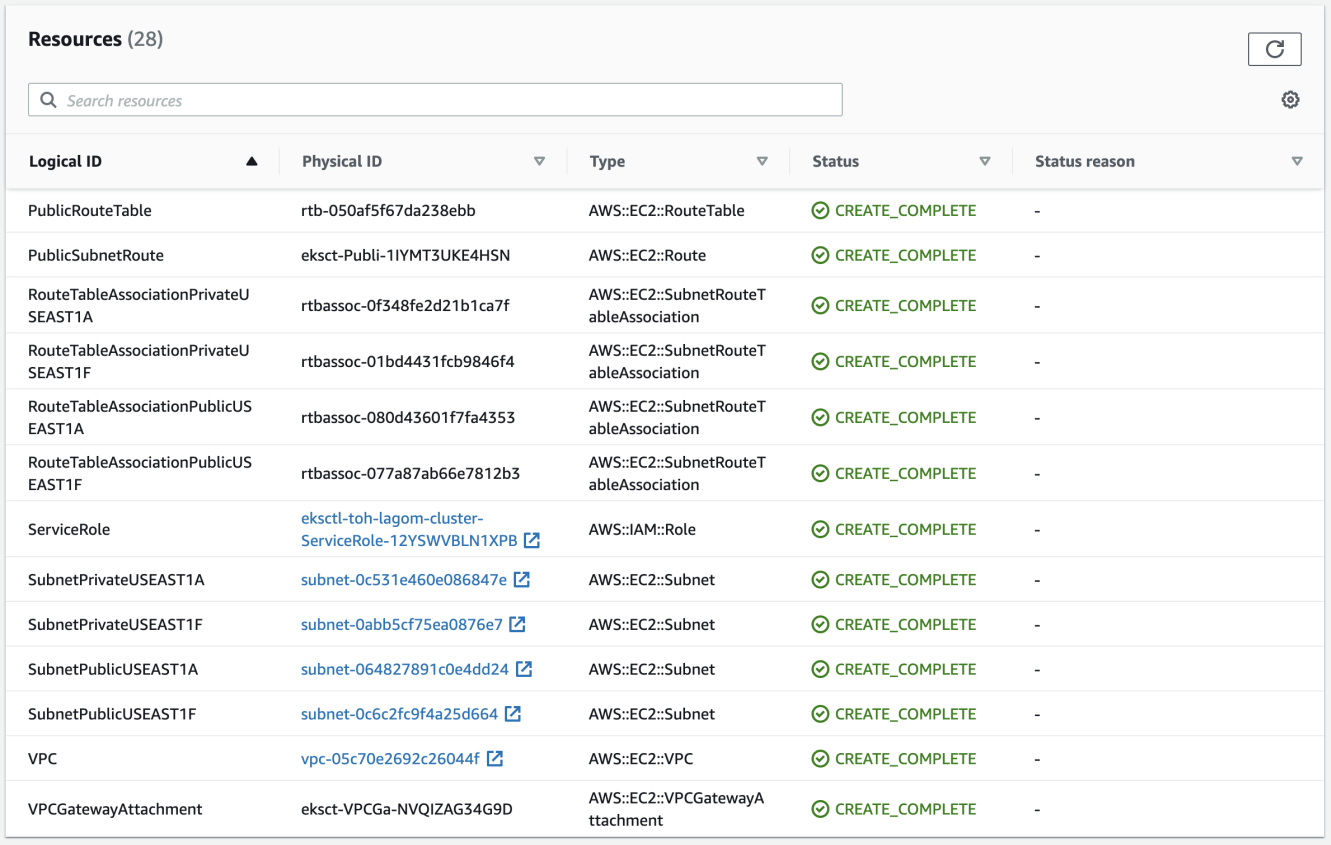
Logical ID	Physical ID	Type	Status	Status reason
ClusterSharedNodeSecurityGroup	sg-0fb9527a468dc35aa	AWS::EC2::SecurityGroup	CREATE_COMPLETE	-
ControlPlane	toh-lagom	AWS::EKS::Cluster	CREATE_COMPLETE	-
ControlPlaneSecurityGroup	sg-0c61e8baec28d2b8d	AWS::EC2::SecurityGroup	CREATE_COMPLETE	-
IngressDefaultClusterToNodeSG	IngressDefaultClusterToNodeSG	AWS::EC2::SecurityGroupIngress	CREATE_COMPLETE	-
IngressInterNodeGroupSG	IngressInterNodeGroupSG	AWS::EC2::SecurityGroupIngress	CREATE_COMPLETE	-
IngressNodeToDefaultClusterSG	IngressNodeToDefaultClusterSG	AWS::EC2::SecurityGroupIngress	CREATE_COMPLETE	-
InternetGateway	igw-079d81c3aa570491d	AWS::EC2::InternetGateway	CREATE_COMPLETE	-
NATGateway	nat-01ee1d0a7cabfceef	AWS::EC2::NatGateway	CREATE_COMPLETE	-
NATIP	52.1.51.114	AWS::EC2::EIP	CREATE_COMPLETE	-
NATPrivateSubnetRouteUSEAST1A	eksct-NATPr-HJUX19F6WUIW	AWS::EC2::Route	CREATE_COMPLETE	-

Figure 6. Cluster Resources



Logical ID	Physical ID	Type	Status	Status reason
NATPrivateSubnetRouteUSEAST1A	eksct-NATPr-HJUX19F6WUIW	AWS::EC2::Route	CREATE_COMPLETE	-
NATPrivateSubnetRouteUSEAST1F	eksct-NATPr-90AZ6P68MXPP	AWS::EC2::Route	CREATE_COMPLETE	-
PolicyCloudWatchMetrics	eksct-Poli-OTAPU4NAQJ9Y	AWS::IAM::Policy	CREATE_COMPLETE	-
PolicyNLB	eksct-Poli-16E5K3W87PS1Q	AWS::IAM::Policy	CREATE_COMPLETE	-
PrivateRouteTableUSEAST1A	rtb-01a99b8b89b9fc44f	AWS::EC2::RouteTable	CREATE_COMPLETE	-
PrivateRouteTableUSEAST1F	rtb-0d0a62c85e57d1194	AWS::EC2::RouteTable	CREATE_COMPLETE	-
PublicRouteTable	rtb-050af5f67da238ebb	AWS::EC2::RouteTable	CREATE_COMPLETE	-
PublicSubnetRoute	eksct-Publi-1IYMT3UKE4HSN	AWS::EC2::Route	CREATE_COMPLETE	-
RouteTableAssociationPrivateUSEAST1A	rtbassoc-0f348fe2d21b1ca7f	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPrivateUSEAST1F	rtbassoc-01bd4431fcb9846f4	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPublicUSEAST1A	rtbassoc-080d43601f7fa4353	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPublicUSEAST1F	rtbassoc-077a87ab66e7812b3	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
ServiceRole	eksctl-toh-lagom-cluster-ServiceRole-12YSWVBLN1XPB	AWS::IAM::Role	CREATE_COMPLETE	-

Figure 7. Cluster Resources, continued



Resources (28)

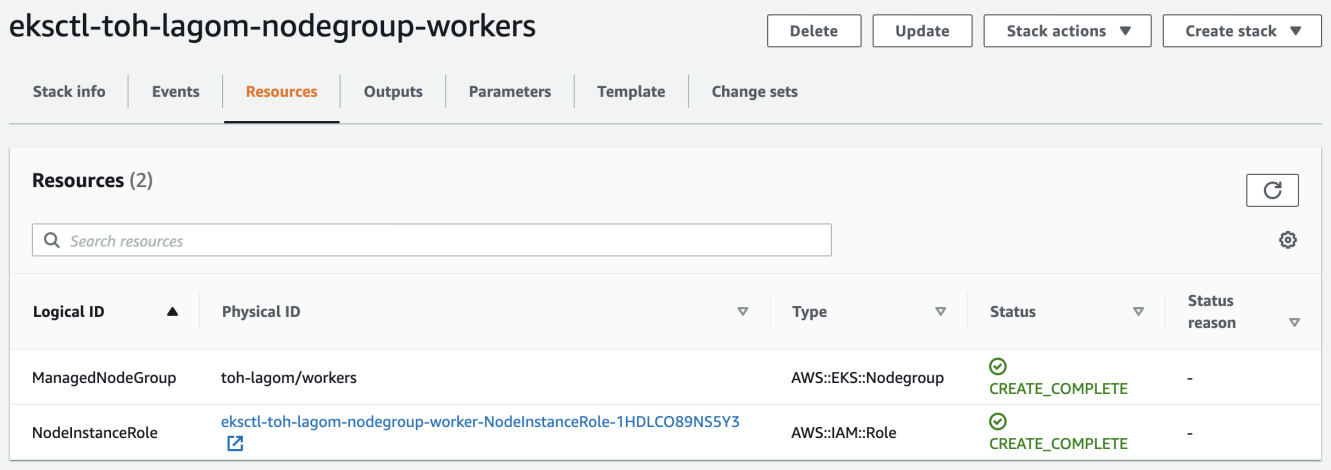
Search resources

Logical ID	Physical ID	Type	Status	Status reason
PublicRouteTable	rtb-050af5f67da238ebb	AWS::EC2::RouteTable	CREATE_COMPLETE	-
PublicSubnetRoute	eksct-Publi-1IYMT3UKE4HSN	AWS::EC2::Route	CREATE_COMPLETE	-
RouteTableAssociationPrivateUSEAST1A	rtbassoc-0f348fe2d21b1ca7f	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPrivateUSEAST1F	rtbassoc-01bd4431fcb9846f4	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPublicUSEAST1A	rtbassoc-080d43601f7fa4353	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
RouteTableAssociationPublicUSEAST1F	rtbassoc-077a87ab66e7812b3	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-
ServiceRole	eksctl-toh-lagom-cluster-ServiceRole-12YSWVBLN1XPB	AWS::IAM::Role	CREATE_COMPLETE	-
SubnetPrivateUSEAST1A	subnet-0c531e460e086847e	AWS::EC2::Subnet	CREATE_COMPLETE	-
SubnetPrivateUSEAST1F	subnet-0abb5cf75ea0876e7	AWS::EC2::Subnet	CREATE_COMPLETE	-
SubnetPublicUSEAST1A	subnet-064827891c0e4dd24	AWS::EC2::Subnet	CREATE_COMPLETE	-
SubnetPublicUSEAST1F	subnet-0c6c2fc9f4a25d664	AWS::EC2::Subnet	CREATE_COMPLETE	-
VPC	vpc-05c70e2692c26044f	AWS::EC2::VPC	CREATE_COMPLETE	-
VPCGatewayAttachment	eksct-VPCGa-NVQIZAG34G9D	AWS::EC2::VPCGatewayAttachment	CREATE_COMPLETE	-

Figure 8. Cluster Resources, continued

Here are the resources created by the second CloudFormation stack, for the Kubernetes managed node group.

If you chose to use unmanaged node groups instead, then the second CloudFormation stack will look as follows.



eksctl-toh-lagom-nodegroup-workers

Delete | Update | Stack actions | Create stack

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Resources (2)

Search resources

Logical ID	Physical ID	Type	Status	Status reason
ManagedNodeGroup	toh-lagom/workers	AWS::EKS::Nodegroup	CREATE_COMPLETE	-
NodeInstanceRole	eksctl-toh-lagom-nodegroup-worker-NodeInstanceRole-1HDLCO89NS5Y3	AWS::IAM::Role	CREATE_COMPLETE	-

Figure 9. Managed Node Group Resources

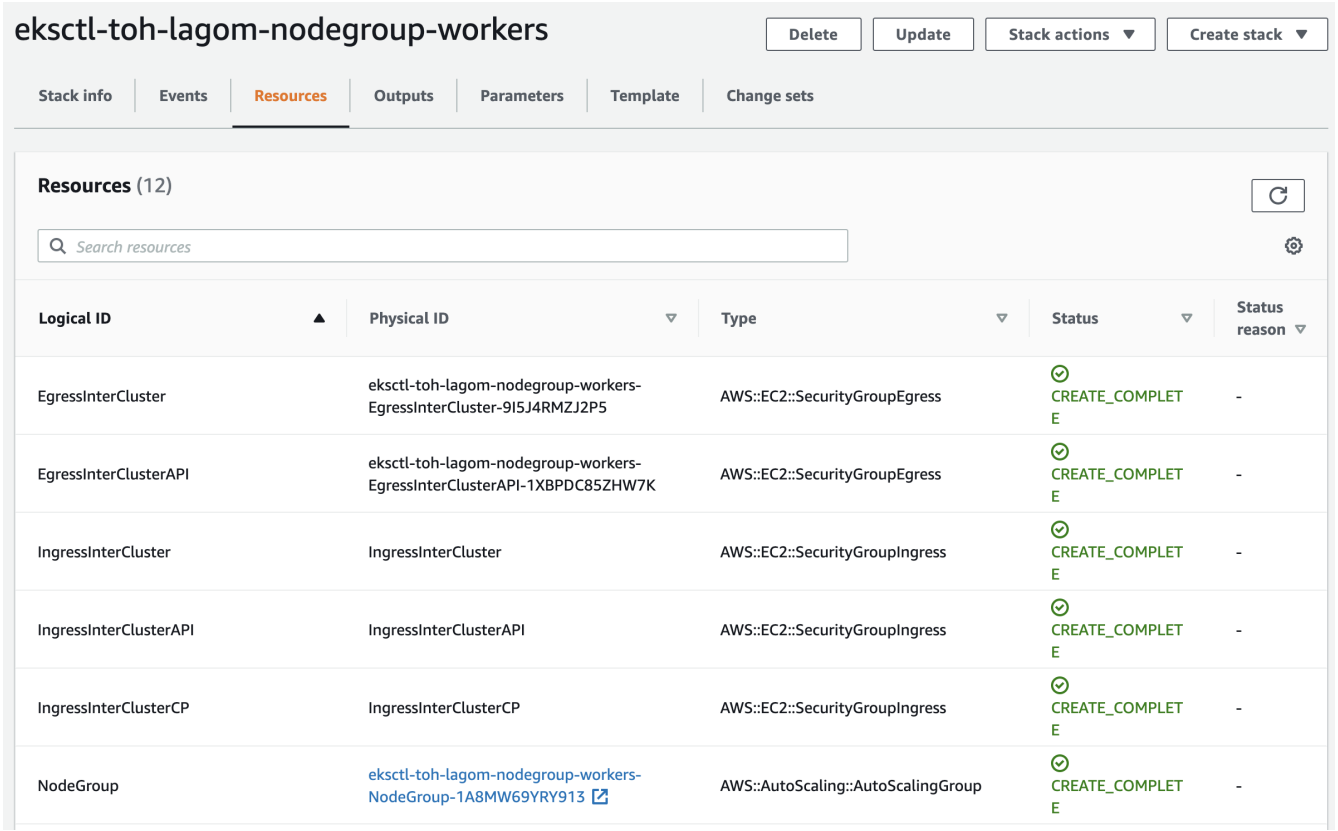


Figure 10. Node Group Resources

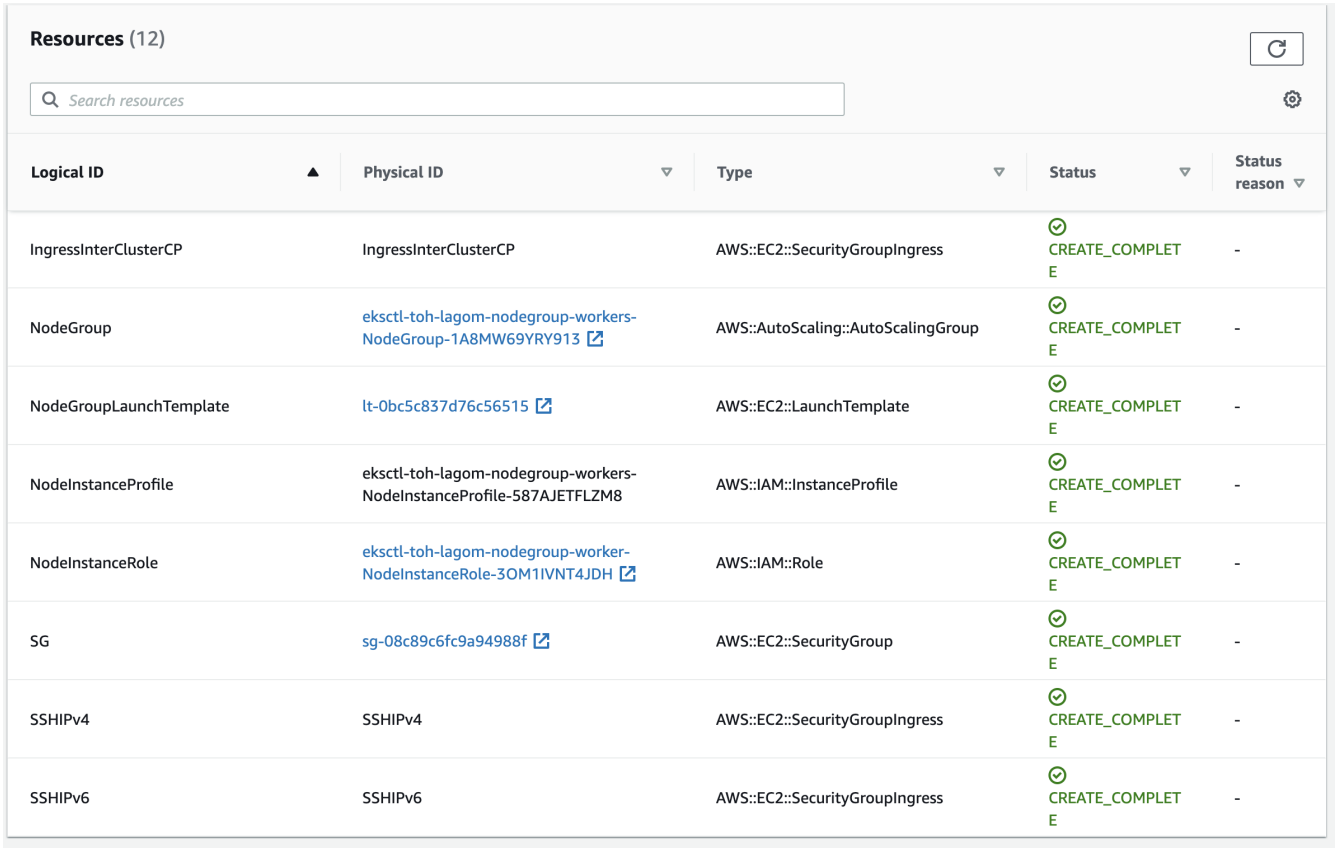


Figure 11. Node Group Resources, continued

Among other things, you'll see that we have created a special Amazon VPC (Virtual Private Cloud) for our cluster. That VPC has one public and one private subnet per each availability zone. Now is the time to create an Aurora database cluster in those private subnets. We also need some information about the created security groups:

deployment/awsdeploy.sh

```
stackInfo=$(aws cloudformation describe-stacks --stack-name eksctl-toh-lagom-cluster)
subnetsPrivate=$(echo ${stackInfo} \
    | jq -r '.Stacks[0].Outputs | .[] | select(.OutputKey == "SubnetsPrivate") |
.OutputValue' \
    | sed 's/,/ /g')
securityGroups=$(echo ${stackInfo} \
    | jq -r '.Stacks[0].Outputs | .[] | select(.OutputKey | endswith("SecurityGroup"))
| .OutputValue')
```

Note the more advanced use of the `jq` utility here. The `select` allows to search the Json output by the values of the fields, not just by their names. This goes beyond the capabilities of AWS CLI's `--query` parameter.

Now, when you run `kubectl get nodes`, you should see two nodes in the `Ready` status, since we haven't started them yet. But we haven't loaded any deployments yet, so `kubectl get deployments` shouldn't show anything.

```
Leonids-MacBook-Pro:deployment lkorogodski$ kubectl get nodes
NAME                                STATUS   ROLES    AGE   VERSION
ip-192-168-28-252.ec2.internal   Ready    <none>   30s   v1.14.9-eks-f459c0
ip-192-168-38-151.ec2.internal   Ready    <none>   29s   v1.14.9-eks-f459c0
```

Now is the time to set up our read-side database.

32. AWS Aurora Serverless

It was important to create an AWS Kubernetes cluster *before* a PostgreSQL database for the read side, because they must reside in the same VPC. Whereas the Amazon RDS CLI allows passing existing subnets as arguments, the `eksctl` utility does not. Now that we have retrieved the private subnets and the associated security groups from the CloudFormation stacks, we can create a database subnet group, which will be necessary for creating the Aurora Serverless cluster.

deployment/awsdeploy.sh

```
echo 'Creating a database subnet group...'
aws rds create-db-subnet-group \
    --db-subnet-group-name toh-lagom-db-subnets \
    --db-subnet-group-description 'Subnets for a database cluster.' \
    --subnet-ids ${subnetsPrivate}
```

Now, we create an Aurora Serverless database cluster in the PostgreSQL mode, passing in the security group IDs retrieved from the CloudFormation stack and the name of the database subnet group that we have just created:

deployment/awsdeploy.sh

```
echo "Creating an Aurora database cluster..."
dbCluster=$(aws rds create-db-cluster \
    --db-cluster-identifier toh-lagom --database-name toh_lagom \
    --engine aurora-postgresql --engine-version 10.7 --engine-mode serverless \
    --master-username ${POSTGRESQL_USERNAME} --master-user-password
${POSTGRESQL_PASSWORD} \
    --scaling-configuration MinCapacity=2,MaxCapacity=8,SecondsUntilAutoPause
=1000,AutoPause=true \
    --vpc-security-group-ids ${securityGroups} --db-subnet-group-name toh-lagom-db-
subnets --enable-http-endpoint)
```

10.7 is the only PostgreSQL version for which there is a serverless option.

Since there is, unfortunately, no `aws rds wait` command for database clusters, we must implement it on our own, in order to wait for the Aurora cluster to become available:

deployment/awsdeploy.sh

```
dbStatus=''
until [[ ${dbStatus} == 'available' ]]
do
    sleep 30
    dbStatus=$(aws rds describe-db-clusters --db-cluster-identifier toh-lagom \
        | jq -r '.DBClusters[0].Status')
    echo "Cluster status: ${dbStatus}..."
done
```

```
echo "Successfully created an Aurora Serverless cluster."
```

Finally, we will query the created AWS Aurora cluster to retrieve the database endpoint, which we will save in the `POSTGRESQL_URL` environment variable. Recall that we use `POSTGRESQL_URL` in our `build.sbt`.

deployment/awsdeploy.sh

```
echo 'Updating the production config with the database data...'
dbEndpoint=$(aws rds describe-db-cluster-endpoints --db-cluster-identifier toh-lagom \
    | jq -r '.DBClusterEndpoints[0].Endpoint')
export POSTGRESQL_URL="jdbc:postgresql://${dbEndpoint}/toh_lagom"
echo "AWS Aurora database: ${POSTGRESQL_URL}"
```

33. Deploying to Kubernetes Cluster

Recall that when we run the deployment script, `deployment/awsdeploy.sh`, we provide a version label as an argument, which is saved in the `TOH_VERSION` environment variable. This label doesn't *have* to have the usual version string format, like `1.0.0`--though it can. If it doesn't reflect any changes in the code but only in the configuration—for example, to build a Docker image for deployment in another AWS region—then it could be something like `1.0.0-us-east-2`.

Next, we can load `k8deployment.yaml` (having substituted the environment variables with their values), run our Kubernetes deployment and autoscale it, then wait for the minimal number of replicas to become available:

deployment/awsdeploy.sh

```
echo 'Starting the Kubernetes service...'
envsubst < k8deployment.yaml | kubectl apply -f -
kubectl autoscale deployment toh-lagom \
    --cpu-percent=50 --min=2 --max=3

readyReplicas=0
until [[ ${readyReplicas} == 2 ]]
do
    sleep 10
    readyReplicas=$(kubectl get deployment toh-lagom -o json \
        | jq '.status.readyReplicas')
    echo "Ready replicas: ${readyReplicas}..."
done
echo 'The Kubernetes service is up and ready.'
```

When executing the deployment script, make sure your Docker daemon is running.

Once that is executed, one can run `kubectl get deployments` in another terminal, while waiting for this to complete. You should see one `toh-lagom` deployment listed. We must wait for both replicas to become available.

```
Leonids-MacBook-Pro:deployment lkorogodski$ kubectl get deployments
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
toh-lagom   0/2     2            0           31s
```

If the script gets stuck on `Ready replicas: null`, run `kubectl get pods` to get the pods' names and then `kubectl logs <pod>` to check if there is an error on startup or whether the pod was able to join the Akka cluster.

In a more complex scenario, with multiple microservices interacting, we could have defined several deployments in the same Kubernetes cluster—and they would have automatically discovered each other when running, if configured with

the Kubernetes API for Akka Discovery as we have done.

You can examine your Kubernetes pods by running `kubectl get pods`:

```
Leonids-MacBook-Pro:deployment lkorogodski$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
toh-lagom-84bc8c4957-brth8    0/1     Running   0          16s
toh-lagom-84bc8c4957-vl86t    0/1     Running   0          16s
```

Then, you can examine the logs of any pod by providing the pod's ID to the following command:

```
kubectl logs [-f] <pod-id>
```

The `-f` option allows you to continue receiving the log messages as an ongoing stream. It is also possible to remotely connect to the pod in order to troubleshoot it in a `bash` shell, by running

```
kubectl exec -it <pod-id> -- /bin/bash
```

If all goes well, the log messages should show that the pod has found the minimal number of peer nodes and has joined the Akka cluster. The log messages should also inform you that the microservice is up and running.

But we must still expose it to the world. The following command creates an Amazon ELB load balancer for the cluster, with a public IP:

deployment/awsdeploy.sh

```
echo 'Exposing the Kubernetes service...'
kubectl expose deployment toh-lagom --type=LoadBalancer --name=toh-lagom-elb
export ELB_ENDPOINT=$(kubectl get service toh-lagom-elb -o json \
    | jq -r '.status.loadBalancer.ingress[0].hostname')

until [[ ${ELB_ENDPOINT} != 'null' ]]
do
    sleep 5
    export ELB_ENDPOINT=$(kubectl get service toh-lagom-elb -o json \
        | jq -r '.status.loadBalancer.ingress[0].hostname')
done
```

The load balancer created automatically by the command above is of the deprecated `Classic` type. At some point, this will likely change.

Here, `elbEndpoint` holds the URL of the load balancer's endpoint, by which our microservice can be accessed from the outside. But we still only serve the restful `/api/*` requests. Next, we will package and deploy the static assets of the Angular's Tour of Heroes tutorial as an AWS CloudFront distribution.

34. Static Website Assets

By the end of the Angular's Tour of Heroes tutorial, you should have a static build of website assets. Alternatively, you can download the final code here. First, however, the following section must be commented out in the frontend code's `app.module.ts` file, in order to disable the in-memory mock of server calls used by the Angular tutorial:

toh-pt6/src/app/app.module.ts

```
    // The HttpClientInMemoryWebApiModule module intercepts HTTP requests
    // and returns simulated server responses.
    // Remove it when a real server is ready to receive requests.
    //HttpClientInMemoryWebApiModule.forRoot(
    //  InMemoryDataService, { dataEncapsulation: false }
    //)
```

When built, the website assets comprise the following files, which we have placed in the `toh-lagom/dist` folder and to which I have added a simple `error.html`:

```
Leonids-MacBook-Pro:toh-lagom lkorogodski$ ls -l dist
total 1640
-rw-r--r--  1 lkorogodski  staff   13418 Apr 16 13:35 3rdpartylicenses.txt
-rw-r--r--  1 lkorogodski  staff     177 Apr 16 13:35 error.html
-rw-r--r--  1 lkorogodski  staff   11736 Apr 16 13:35 favicon.ico
-rw-r--r--  1 lkorogodski  staff     809 Apr 16 13:35 index.html
-rw-r--r--  1 lkorogodski  staff  289355 Apr 16 13:35 main-
es2015.290d141d38cb82b91d31.js
-rw-r--r--  1 lkorogodski  staff  338991 Apr 16 13:35 main-es5.26f5fbf052010bedabc7.js
-rw-r--r--  1 lkorogodski  staff   37160 Apr 16 13:35 polyfills-
es2015.665667c61913a16988b5.js
-rw-r--r--  1 lkorogodski  staff  114891 Apr 16 13:35 polyfills-
es5.e02bc17ce8b8e6b9ed8a.js
-rw-r--r--  1 lkorogodski  staff    1440 Apr 16 13:35 runtime-
es2015.858f8dd898b75fe86926.js
-rw-r--r--  1 lkorogodski  staff    1440 Apr 16 13:35 runtime-
es5.741402d1d47331ce975c.js
-rw-r--r--  1 lkorogodski  staff     765 Apr 16 13:35 styles.720f960b4e8d5bebe737.css
```

Recall that our deployment script takes an optional second command line argument, which serves as the name of the S3 bucket in which the above assets are stored. You can create it separately beforehand and copy all these files to it. However, if the S3 bucket's name is not provided, it will be created by the script.

This S3 bucket will serve us as storage for the CloudFront distribution we are building. Because S3 bucket names must be globally unique, we will append a random suffix to the bucket's name. Then, we create the S3 bucket, wait until it is ready, and copy the static assets from `dist` to the bucket.

deployment/awsdeploy.sh

```
bucket=$2
if [[ -z ${TOH_VERSION} ]]
then
    export LC_CTYPE=C
    suffix=$(head /dev/urandom | tr -dc a-z0-9 | head -c 13)
    bucket="toh-lagom-${suffix}"

    echo "Creating S3 bucket ${bucket} to serve as a static website..."
    aws s3 mb "s3://${bucket}" --region ${AWS_REGION}
    aws s3api wait bucket-exists --bucket ${bucket}
    aws s3 cp ../dist "s3://${bucket}" --recursive --storage-class INTELLIGENT_TIERING
else
    echo "Using S3 bucket ${bucket} to serve as a static website..."
fi
```

Before we create an AWS CloudFront distribution, we must create an AWS Origin Access Identity and grant it the read access to our S3 bucket.

deployment/awsdeploy.sh

```
echo 'Creating a CloudFront origin access identity...'
originAccessIdentity=$(aws cloudfront create-cloud-front-origin-access-identity \
    --cloud-front-origin-access-identity-config CallerReference=toh-lagom,Comment=toh-
lagom \
    | jq -r '.CloudFrontOriginAccessIdentity.Id')
sleep 10
```

The wait is necessary, because the origin access identity is not immediately available for use as a principal in an S3 bucket's permissions policy or in a CloudFront distribution, even though the AWS resource itself exists. Unfortunately, there is no handy way to check if it's available, so we just simply sleep for a few seconds in the script.

In order to configure an S3 permissions policy, create an `s3policy.json` file in the `deployment` folder, with the following contents:

deployment/s3policy.json

```
{
  "Version": "2012-10-17",
  "Id": "PolicyForCloudFrontPrivateContent",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "${S3_POLICY_PRINCIPAL}"
      },
      "Action": [
        "s3:GetObject"
```

```
      ],
      "Resource": "${S3_POLICY_RESOURCE}"
    }
  ]
}
```

We substitute the environment variables' values by means of the `envsubst` command line utility and apply the policy to the S3 bucket.

deployment/awsdeploy.sh

```
echo 'Authorizing the CloudFront origin access identity to read from the S3 bucket...'
export S3_POLICY_RESOURCE="arn:aws:s3:::${bucket}/*"
export S3_POLICY_PRINCIPAL="arn:aws:iam::cloudfront:user/CloudFront Origin Access
Identity ${originAccessIdentity}"
envsubst < s3policy.json > policy.json
aws s3api put-bucket-policy --bucket ${bucket} --policy file://policy.json
rm -rf policy.json
```

35. CloudFront Distribution

AWS CloudFront distributions are great for serving static websites fast. But it is also possible to use them to serve static assets for the websites that also must make server calls. This is done by adding non-default behaviors to a CloudFront fistribution.

In our case, we will create a CloudFront distribution with the default behavior to serve files from the S3 bucket into which we have placed our static assets. That distrubition will also have a non-default behavior that redirects all `/api/*` server calls to the AWS load balancer that we created to expose an endpoint to our AWS Kubernetes cluster.

Create a `cloudfront.json` file in the `deployment` folder to serve as the configuration for our CloudFront distribution.

deployment/cloudfront.json

```
{
  "CallerReference": "toh-lagom",
  "Aliases": {
    "Quantity": 0
  },
  "DefaultRootObject": "index.html",
  "Origins": {
    "Quantity": 2,
    "Items": [
      {
        "Id": "s3-toh-lagom",
        "DomainName": "${S3_ENDPOINT}",
        "OriginPath": "",
        "CustomHeaders": {
          "Quantity": 0
        },
        "S3OriginConfig": {
          "OriginAccessIdentity": "${ORIGIN_ACCESS_IDENTITY}"
        }
      },
      {
        "Id": "elb-toh-lagom",
        "DomainName": "${ELB_ENDPOINT}",
        "OriginPath": "",
        "CustomHeaders": {
          "Quantity": 0
        },
        "CustomOriginConfig": {
          "HTTPPort": 9000,
          "HTTPSPort": 9000,
          "OriginProtocolPolicy": "match-viewer",
          "OriginSslProtocols": {
            "Quantity": 4,
            "Items": ["SSLv3", "TLSv1", "TLSv1.1", "TLSv1.2"]
```

```
        },
        "OriginReadTimeout": 30,
        "OriginKeepaliveTimeout": 5
      }
    }
  ]
},
"OriginGroups": {
  "Quantity": 0
},
"DefaultCacheBehavior": {
  "TargetOriginId": "s3-toh-lagom",
  "ForwardedValues": {
    "QueryString": false,
    "Cookies": {
      "Forward": "none"
    },
    "Headers": {
      "Quantity": 0
    },
    "QueryStringCacheKeys": {
      "Quantity": 0
    }
  },
  "TrustedSigners": {
    "Enabled": false,
    "Quantity": 0
  },
  "ViewerProtocolPolicy": "allow-all",
  "MinTTL": 0,
  "AllowedMethods": {
    "Quantity": 2,
    "Items": ["GET", "HEAD"],
    "CachedMethods": {
      "Quantity": 2,
      "Items": ["GET", "HEAD"]
    }
  },
  "SmoothStreaming": false,
  "DefaultTTL": 86400,
  "MaxTTL": 31536000,
  "Compress": false,
  "LambdaFunctionAssociations": {
    "Quantity": 0
  },
  "FieldLevelEncryptionId": ""
},
"CacheBehaviors": {
  "Quantity": 1,
  "Items": [
    {
```

```
        "PathPattern": "api/*",
        "TargetOriginId": "elb-toh-lagom",
        "ForwardedValues": {
          "QueryString": true,
          "Cookies": {
            "Forward": "none",
            "WhitelistedNames": {
              "Quantity": 0
            }
          },
          "Headers": {
            "Quantity": 0
          },
          "QueryStringCacheKeys": {
            "Quantity": 1,
            "Items": ["name"]
          }
        },
        "TrustedSigners": {
          "Enabled": false,
          "Quantity": 0
        },
        "ViewerProtocolPolicy": "allow-all",
        "MinTTL": 0,
        "AllowedMethods": {
          "Quantity": 7,
          "Items": ["HEAD", "DELETE", "POST", "GET", "OPTIONS", "PUT", "PATCH"],
          "CachedMethods": {
            "Quantity": 2,
            "Items": ["GET", "HEAD"]
          }
        },
        "SmoothStreaming": false,
        "DefaultTTL": 10,
        "MaxTTL": 300,
        "Compress": false,
        "LambdaFunctionAssociations": {
          "Quantity": 0
        },
        "FieldLevelEncryptionId": ""
      }
    ]
  },
  "CustomErrorResponses": {
    "Quantity": 0
  },
  "Comment": "Tour of Heroes - Lagom Backend",
  "Logging": {
    "Enabled": false,
    "IncludeCookies": false,
    "Bucket": "",
```

```
    "Prefix": ""
  },
  "PriceClass": "PriceClass_100",
  "Enabled": true,
  "ViewerCertificate": {
    "CloudFrontDefaultCertificate": true,
    "MinimumProtocolVersion": "TLSv1",
    "CertificateSource": "cloudfront"
  },
  "Restrictions": {
    "GeoRestriction": {
      "RestrictionType": "none",
      "Quantity": 0
    }
  },
  "WebACLId": "",
  "HttpVersion": "http2",
  "IsIPV6Enabled": true
}
```

Here, we define two origins, `s3-toh-lagom` and `elb-toh-lagom`, for the static assets in the S3 bucket and for our load balancer respectively. The default behavior uses the former, while another behavior forwards all requests that match the `api/*` path pattern to the load balancer. Here is how the behaviors look in the AWS Console when the CloudFront distribution is created.

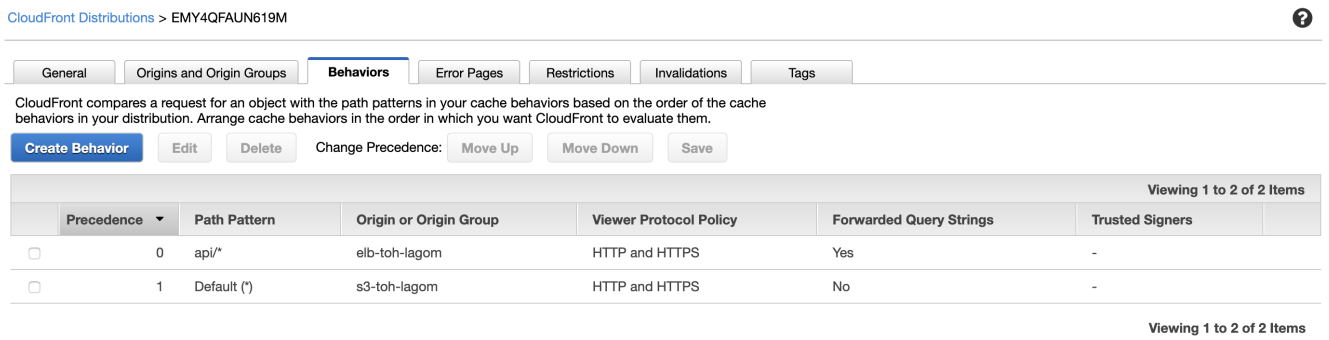


Figure 12. CloudFront Behaviors

Again, we use the `envsubst` command line utility to substitute the environment variables' values for the S3 and ELB endpoints and for the Origin Access Identity. Then, we create an AWS CloudFront distribution using that configuration. Finally, we must wait for the CloudFront distribution to deploy.

deployment/awsdeploy.sh

```
echo 'Creating a CloudFront distribution...'
export S3_ENDPOINT="${bucket}.s3.amazonaws.com"
export ORIGIN_ACCESS_IDENTITY="origin-access-identity/cloudfront/
${originAccessIdentity}"
envsubst < cloudfront.json > distribution.json
distributionId=$(aws cloudfront create-distribution --distribution-config
file://distribution.json \
```

```
    | jq -r '.Distribution.Id')
rm -rf distribution.json

echo 'Waiting for the CloudFront distribution to deploy...'
aws cloudfront wait distribution-deployed --id ${distributionId}
export CF_ENDPOINT=$(aws cloudfront get-distribution --id ${distributionId} \
    | jq -r '.Distribution.DomainName')
echo "Successfully deployed a Cloudfront distribution: ${CF_ENDPOINT}"

unset AWS_MCS_USERNAME
unset AWS_MCS_PASSWORD
unset POSTGRESQL_USERNAME
unset POSTGRESQL_PASSWORD
unset cassandraSecret
unset dbSecret
```

Once the wait is over, we have our publicly accessible URL in the `CF_ENDPOINT` environment variable. Finally, we unset the sensitive environment variables. The website is ready to serve content!

36. Upgrade Script

The upgrade script `deployment/awsupgrade.sh` is designed to upgrade a running AWS Kubernetes cluster with another Docker image, with automatic restart. It does not affect any other AWS resources created by the deployment script `deployment/awsdeploy.sh`.

The upgrade script also takes a version label as an argument. It also saves it to the `TOH_VERSION` environment variables and checks whether the ECR repository already has a Docker image with that label. If not, it builds one, then waits until it is available in the repository.

deployment/awsupgrade.sh

```
#!/usr/bin/env bash

export TOH_VERSION=$1
if [[ -z ${TOH_VERSION} ]]
then
    echo 'Must provide a version.'
    exit 1
else
    echo "Using the toh-lagom version ${TOH_VERSION}."
fi

repositoryUri=$(aws ecr describe-repositories --repository-names toh-lagom \
    | jq -r '.repositories[0].repositoryUri')
echo "AWS ECR repository: ${repositoryUri}"

if [[ -z $(aws ecr describe-images --repository-name toh-lagom --image-ids imageTag=
${TOH_VERSION}) ]]
then
    echo "The Docker image with version ${TOH_VERSION} is NOT found."
    echo 'Building the ECR Docker image...'
    cd ..; sbt clean ecr:push; cd ./deployment
    echo 'Pushed the Docker image into AWS ECR.'
else
    echo "The Docker image with version ${TOH_VERSION} is found."
fi

until [[ -n $(aws ecr describe-images --repository-name toh-lagom --image-ids
imageTag=${TOH_VERSION}) ]]
do
    sleep 5
done

echo 'Configuring Kube config to toh-lagom...'
aws eks update-kubeconfig --name toh-lagom
kubectl set image deployment.apps/toh-lagom toh-lagom=${repositoryUri}:${TOH_VERSION}
echo "Upgraded the Docker image in the Kubernetes cluster to ${TOH_VERSION}."
```

The last few lines upgrade the AWS Kubernetes cluster. If the same user ran the deployment script—and therefore, the `eksctl create cluster` command—from the same machine, then the Kube configuration should already point to the cluster. But if not, then the `aws eks update-kubeconfig command sets the Kube configuration properly. Then, we simply update the Docker image, which restarts the cluster's Kubernetes pods.

Upgrading our cluster then becomes as easy as running this from the `deployment` folder:

```
. awsupgrade.sh <version>
```

We will have a chance to do that once we implement a workaround for an issue we discuss in the next chapter.

37. Cleanup Script

The following script deletes all resources created by the deployment script, except for the S3 bucket and the ECR repository with its versioned Docker images. We do it in the opposite order, starting with the CloudFront distribution. We find it by filtering all distributions by the comment, which we have set to `Tour of Heroes - Lagom Backend` in the `deployment/cloudfront.json` template. Before it can be deleted, however, it must be disabled; otherwise, AWS CLI returns an error. In order to disable the distribution, we must obtain the `ETag` from its previous update, then retrieve its current config, save it into a temporary file, replace the `Enabled` field in the config with `false`, and update the distribution with that config. Then, we wait for the update to propagate and delete the distribution.

deployment/awscleanup.sh

```
#!/usr/bin/env bash

echo 'Disabling the CloudFront distribution...'
distributionId=$(aws cloudfront list-distributions \
    | jq -r '.DistributionList.Items | .[] | select(.Comment == "Tour of Heroes -
Lagom Backend") | .Id')
distributionEtag=$(aws cloudfront get-distribution --id ${distributionId} | jq -r
'.ETag')
aws cloudfront get-distribution-config --id ${distributionId} \
    | jq -r '.DistributionConfig' | jq '.Enabled=false' > config.json
distribution=$(aws cloudfront update-distribution --id ${distributionId} \
    --if-match ${distributionEtag} --distribution-config file://config.json)
aws cloudfront wait distribution-deployed --id ${distributionId}
rm -rf config.json

echo 'Deleting the CloudFront distribution...'
distributionEtag=$(aws cloudfront get-distribution --id ${distributionId} | jq -r
'.ETag')
aws cloudfront delete-distribution --id ${distributionId} --if-match
${distributionEtag}
```

Then, we delete the CloudFront origin access identity, finding it by its comment, which we set to `toh-lagom`. The script doesn't remove the origin access identity from the S3 bucket's policy, though. That can be done manually; although, since the origin access identity no longer exists, it is safe to leave it in place.

deployment/awscleanup.sh

```
echo 'Deleting the CloudFront Origin Access Identity...'
originAccessIdentity=$(aws cloudfront list-cloud-front-origin-access-identities \
    | jq -r '.CloudFrontOriginAccessIdentityList.Items | .[] | select(.Comment ==
"toh-lagom") | .Id')
oidEtag=$(aws cloudfront get-cloud-front-origin-access-identity --id
${originAccessIdentity} \
    | jq -r '.ETag')
```

```
aws cloudfront delete-cloud-front-origin-access-identity \
    --id ${originAccessIdentity} --if-match ${oidEtag}
```

Recall that when we called the following command in the deployment script, a load balancer was created.

deployment/awsdeploy.sh

```
kubectl expose deployment toh-lagom --type=LoadBalancer --name=toh-lagom-elb
```

We must delete it before the Kubernetes cluster. Otherwise, if we try deleting the cluster first, the `elsctl` command line utility will return an error, complaining that some resources cannot be deleted. This is because creation of the load balancer modified the cluster resources, which now have dependents outside of the cluster.

In order to delete the load balancer, we must find the cluster's VPC, which we do by examining the cluster's resources. Then, we look for the only load balancer in that VPC. However, it is not enough to just delete that load balancer. We must also delete its security group, which was created along with it. Finding it is easy (see below), but we cannot delete it yet, because it was added as a source in an ingress permission for one of the security groups in the Kubernetes cluster (this is where the dependency relationship comes from). So, we must search for the affected security group and revoke the ingress permission that uses the load balancer's security group.

deployment/awscleanup.sh

```
echo 'Deleting the load balancer...'
vpcId=$(aws cloudformation describe-stacks --stack-name eksctl-toh-lagom-cluster \
    | jq -r '.Stacks[0].Outputs | .[] | select(.OutputKey == "VPC") | .OutputValue')
elbInfo=$(aws elb describe-load-balancers \
    | jq -r --arg vpc ${vpcId} '.LoadBalancerDescriptions | .[] | select(.VPCId ==
$vpc)')
elbName=$(echo ${elbInfo} | jq -r '.LoadBalancerName')
elbSecurityGroup=$(echo ${elbInfo} | jq -r '.SecurityGroups[0]')
dependentSecurityGroup=$(aws ec2 describe-security-groups \
    --filters Name=ip-permission.group-id,Values=${elbSecurityGroup} \
    | jq -r '.SecurityGroups[0].GroupId')
aws elb delete-load-balancer --load-balancer-name ${elbName}
aws ec2 revoke-security-group-ingress --group-id ${dependentSecurityGroup} \
    --ip-permissions IpProtocol=-1,UserIdGroupPairs=[{GroupId=${elbSecurityGroup}}]
```

Still, we cannot delete the load balancer's security group, because we must wait for the requested changes to take effect. In the meantime, we can delete the Aurora database cluster, which should take long enough.

deployment/awscleanup.sh

```
echo 'Deleting the Aurora database cluster...'
until [[ -z $(aws rds delete-db-cluster --db-cluster-identifier toh-lagom --skip-final
-snapshot) ]]
```

```
do
    sleep 10
done
aws rds delete-db-subnet-group --db-subnet-group-name toh-lagom-db-subnets
```

Now, at last, we can delete the load balancer's security group. Then, we invoke the `eksctl delete cluster` command to delete the Kubernetes cluster. The last command takes care of everything in both of the Kubernetes cluster's CloudFormation stacks. However, if we had tried executing that command too soon, we would have ended up with an error and an incompletely cleaned up state.

deployment/awscleanup.sh

```
echo 'Deleting the Kubernetes cluster...'
aws ec2 delete-security-group --group-id ${elbSecurityGroup}
eksctl delete cluster --region=${AWS_REGION} --name=toh-lagom --wait
```

Part VII. Additional Considerations

38. Troubleshooting

You may recall how we developed a custom session provider, `AmazonSessionProvider`, in order to be able to reconnect to AWS Cassandra when its sessions become stale. Unfortunately, if you let you mnicroservice running long enough, you will notice that the issue persists.

This problem is caused by a disagreement between the versions of DataStax Java driver for Cassandra and Lagom that we are using here. If you use different versions, you may or may not see the same problem.

Lagom's default setting for the connection pool timeout is zero. However, the DataStax driver throws a `BusyPoolException` if the timeout is zero:

```
    if (timeout == 0 || maxQueueSize == 0) {
      return Futures.immediateFailedFuture(new BusyPoolException(host.
getSocketAddress(), 0));
    }
```

The workaround is to set the connection pool timeout to the DataStax default of 5000 milliseconds. The final version of `cassandra.conf` should look as follows:

toh-lagom-impl/src/main/resources/cassandra.conf

```
cassandra.default {
  keyspace = toh_lagom_dev
  # DataStax Cassandra driver requires a non-zero value.
  connection-pool.pool-timeout-millis = 5000

  # AWS Cassandra doesn't currently support QUORUM.
  read-consistency = "LOCAL_QUORUM"
  write-consistency = "LOCAL_QUORUM"

  ssl.truststore {
    path = "cassandra_truststore.jks"
    password = "amazon"
  }

  authentication {
    username = ${AWS_MCS_USERNAME}
    password = ${AWS_MCS_PASSWORD}
  }
}

cassandra-journal {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = ${cassandra.default.read-consistency}
  write-consistency = ${cassandra.default.write-consistency}
  connection-pool.pool-timeout-millis = ${cassandra.default.connection-pool.pool-
timeout-millis}
```

```
  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }

  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
  }
}

cassandra-query-journal {
  read-consistency = ${cassandra.default.read-consistency}
  connection-pool.pool-timeout-millis = ${cassandra.default.connection-pool.pool-
timeout-millis}
}

cassandra-snapshot-store {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = "ONE"
  write-consistency = "ONE"
  connection-pool.pool-timeout-millis = ${cassandra.default.connection-pool.pool-
timeout-millis}

  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }

  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
  }
}

lagom.persistence.read-side.cassandra {
  keyspace = ${cassandra.default.keyspace}
  read-consistency = ${cassandra.default.read-consistency}
  write-consistency = ${cassandra.default.write-consistency}
  connection-pool.pool-timeout-millis = ${cassandra.default.connection-pool.pool-
timeout-millis}

  ssl.truststore {
    path = ${cassandra.default.ssl.truststore.path}
    password = ${cassandra.default.ssl.truststore.password}
  }

  authentication {
    username = ${cassandra.default.authentication.username}
    password = ${cassandra.default.authentication.password}
```

```
  }
}
```

Following this change, you'll notice that the microservice does reconnect to AWS Cassandra. However, the requests during which a stale connection is discovered fail and aren't automatically retried in the background, neither by Lagom nor by the DataStax driver.

Unfortunately, this is not easy to work around. So, until either Lagom or DataStax comes up with a fix, we will have to tolerate an occasional failure upon encountering a slate connection, one per running Kubernetes pod, because each must reconnect.

Should the Cassandra contact points go stale, one can perform a rolling restart of the Kubernetes pods, as follows:

```
kubectl rollout restart deployment/toh-lagom
```

For Kubernetes-related problems, refer to the AWS Kubernetes Troubleshooting Guide and the Kubernetes Cheat Sheet.

A copy of the default `logback.xml` file is provided in GitHub, in the `toh-lagom-impl/src/main/resources` folder, for your convenience, should you need to troubleshoot other issues. In particular, it includes the logging settings for the DataStax driver.

39. Akka Split Brain Resolver

When running multiple nodes in an Akka cluster, one must have a strategy for what to do when a group of nodes become disconnected from the rest of the cluster. This can lead to very subtle, complex issues that are difficult to resolve.

The main complication is that, to any given node, the loss of connection to another is indistinguishable from a crash. However, the disconnected node(s) can continue serving requests and thus updating the journal—and, through event providers or via Kafka, the read side, as well. So, this cannot be handled by simply downing the disconnected nodes and restarting them. For one thing, how does one decide which of the disconnected groups of nodes are to be downed? For another, different nodes may now have different views of the write and read sides, so the problem is in how to resolve any discrepancies.

Lightbend provides a library called Akka Split Brain Resolver that addresses them. Its documentation provides a very good description of the problem and several strategies that can be adopted, depending on the circumstances. We won't repeat it here.

However, Akka Split Brain Resolver is a proprietary library, requiring a paid subscription to the Lightbend platform. So, we won't use it for our toy microservice.

40. Java Implementation

The Java version of the same Lagom microservice can be found here. We won't go over it in the same step-by-step detail, because in many aspects it is very similar. We will highlight, however, several important differences, which fall into two categories:

- Json (de)serialization in Java.
- Runtime dependency injection with Guice.
- Using JUnit 4 for testing.

The Play Json library works a little differently in Java than it does in Scala. Instead of providing an implicit instance of `Format[T]` to be in scope, it relies on Jackson annotation on the classes, constructors, and constructor arguments. Here, for example, is how the `Hero` class would be defined—in its own file, this time.

toh-lagom-java-api/src/main/java/com/chariotsolutions/tohlagom/api/Hero.java

```
package com.chariotsolutions.tohlagom.api;

import lombok.Value;
import com.google.common.base.Preconditions;
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;

@Value
@JsonDeserialize
public final class Hero {
    public final String id;
    public final String name;

    @JsonCreator
    public Hero(@JsonProperty("id") String id,
                @JsonProperty("name") String name) {
        this.id = Preconditions.checkNotNull(id, "id");
        this.name = Preconditions.checkNotNull(name, "name");
    }
}
```

The other classes in need of Json (de)serialization would look similar, with the same familiar—but unfortunate—Java boilerplate code. This includes the `HeroState` class; the `Confirmation` interface, with `Accepted` and `Rejected` as nested classes; and the `HeroEvent` interface, with `HeroCreated`, `HeroChanged`, and `HeroDeleted` as nested classes. They must also implement the Lagom's `Jsonable` interface. For example:

toh-lagom-java-api/src/main/java/com/chariotsolutions/tohlagom/impl/HeroEvent.java

```
package com.chariotsolutions.tohlagom.impl;
```

```
import lombok.Value;
import com.google.common.base.Preconditions;
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.lightbend.lagom.javadsl.persistence.AggregateEventTagger;
import com.lightbend.lagom.javadsl.persistence.AggregateEventTag;
import com.lightbend.lagom.javadsl.persistence.AggregateEvent;
import com.lightbend.lagom.serialization.Jsonable;

public interface HeroEvent extends Jsonable, AggregateEvent<HeroEvent> {
    int NUM_SHARDS = 2;
    AggregateEventTagger<HeroEvent> TAG = NUM_SHARDS > 1
            ? AggregateEventTag.sharded(HeroEvent.class, NUM_SHARDS)
            : AggregateEventTag.of(HeroEvent.class);

    @Override
    default AggregateEventTagger<HeroEvent> aggregateTag() {
        return TAG;
    }

    @Value
    @JsonDeserialize
    final class HeroCreated implements HeroEvent {
        public final String id;
        public final String name;

        @JsonCreator
        HeroCreated(@JsonProperty("id") String id,
                    @JsonProperty("name") String name) {
            this.id = Preconditions.checkNotNull(id, "id");
            this.name = Preconditions.checkNotNull(name, "name");
        }
    }

    @Value
    @JsonDeserialize
    final class HeroChanged implements HeroEvent {
        public final String id;
        public final String newName;
        public final String oldName;

        @JsonCreator
        HeroChanged(@JsonProperty("id") String id,
                    @JsonProperty("newName") String newName,
                    @JsonProperty("oldName") String oldName) {
            this.id = Preconditions.checkNotNull(id, "id");
            this.newName = Preconditions.checkNotNull(newName, "newName");
            this.oldName = Preconditions.checkNotNull(oldName, "oldName");
        }
```

```
    }

    @Value
    @JsonDeserialize
    final class HeroDeleted implements HeroEvent {
        public final String id;

        @JsonCreator
        HeroDeleted(@JsonProperty("id") String id) {
            this.id = Preconditions.checkNotNull(id, "id");
        }
    }
}
```

But `HeroCommand` is (de)serialized with Akka Jackson.

The `@Value` annotation comes from the Lombok library and marks the class as being immutable. The library must be added as a dependency in `build.sbt`, of course: `val lombok = "org.projectlombok" % "lombok" % "1.18.8"`

41. Java Runtime Dependency Injection

In the Scala version, we use compile-time dependency injection, accomplished by mixing in such traits as `WriteSideCassandraPersistenceComponents` and `ReadSideJdbcPersistenceComponents` into `TourOfHeroesApplication` that is loaded by `TourOfHeroesLoader`, which further mixes in `AkkaDiscoveryComponents` in production and `LagomDevModeComponents` in development.

In the Java version, we use Guice for runtime dependency injection. Instead of application loaders, we work with Guice modules. Here is how the Cassandra-only module looks:

toh-lagom-java-impl/src/main/java/com/chariotsolutions/tohlagom/cassandra/TourOfHeroesModule.java

```
package com.chariotsolutions.tohlagom.cassandra;

import play.Environment;
import com.typesafe.config.Config;
import com.chariotsolutions.tohlagom.api.TourOfHeroesService;

public class TourOfHeroesModule extends com.chariotsolutions.tohlagom.common
.TourOfHeroesModule {
    public TourOfHeroesModule(Environment environment, Config config) {
        super(environment);
    }

    @Override
    protected void configure() {
        super.configure();
        bindService(TourOfHeroesService.class, TourOfHeroesServiceImpl.class);
    }
}
```

The module must be enabled in `application.conf`, as we shall see below. For the mixed mode, however, it is not enough to bind the `TourOfHeroesService` class. One must also disable the `JdbcPersistenceModule` module in `application.conf`.

toh-lagom-java-impl/src/main/resources/application.conf

```
include "cassandra"
include "postgresql"

play.modules.enabled += com.chariotsolutions.tohlagom.mixed.TourOfHeroesModule
play.modules.disabled +=
com.lightbend.lagom.javadsl.persistence.jdbc.JdbcPersistenceModule

akka {
  actor.serialization-bindings {
    "com.chariotsolutions.tohlagom.impl.HeroCommand" = jackson-json
  }

  persistence {
```

```
    journal.plugin = cassandra-journal
    snapshot-store.plugin = cassandra-snapshot-store
  }
}
```

This requires to make some JDBC-related bindings explicitly in the mixed-mode module.

toh-lagom-java-impl/src/main/java/com/chariotsolutions/tohlagom/mixed/TourOfHeroesModule.java

```
package com.chariotsolutions.tohlagom.mixed;

import play.Environment;
import com.typesafe.config.Config;
import com.lightbend.lagom.javadsl.persistence.jdbc.JdbcSession;
import com.lightbend.lagom.javadsl.persistence.jdbc.JdbcReadSide;
import com.lightbend.lagom.internal.persistence.jdbc.SlickOffsetStore;
import com.lightbend.lagom.javadsl.persistence.jdbc.GuiceSlickProvider;
import com.lightbend.lagom.internal.javadsl.persistence.jdbc.SlickProvider;
import com.lightbend.lagom.internal.javadsl.persistence.jdbc.JdbcSessionImpl;
import com.lightbend.lagom.internal.javadsl.persistence.jdbc.JdbcReadSideImpl;
import com.lightbend.lagom.internal.javadsl.persistence.jdbc.JavadslJdbcOffsetStore;
import com.chariotsolutions.tohlagom.api.TourOfHeroesService;

public class TourOfHeroesModule extends com.chariotsolutions.tohlagom.common
.TourOfHeroesModule {
    public TourOfHeroesModule(Environment environment, Config config) {
        super(environment);
    }

    @Override
    protected void configure() {
        super.configure();
        bindService(TourOfHeroesService.class, TourOfHeroesServiceImpl.class);
        // JdbcPersistenceModule is disabled in application.conf to
        // avoid conflicts with CassandraPersistenceModule.
        bind(SlickProvider.class).toProvider(GuiceSlickProvider.class);
        bind(SlickOffsetStore.class).to(JavadslJdbcOffsetStore.class);
        bind(JdbcReadSide.class).to(JdbcReadSideImpl.class);
        bind(JdbcSession.class).to(JdbcSessionImpl.class);
    }
}
```

The common base module binds the service locator in the production mode.

toh-lagom-java-impl/src/main/java/com/chariotsolutions/tohlagom/common/TourOfHeroesModule.java

```
package com.chariotsolutions.tohlagom.common;

import play.Environment;
import com.google.inject.AbstractModule;
```

```
import com.lightbend.lagom.javadsl.api.ServiceLocator;
import com.lightbend.lagom.javadsl.server.ServiceGuiceSupport;
import com.lightbend.lagom.javadsl.akka.discovery.AkkaDiscoveryServiceLocator;

public class TourOfHeroesModule extends AbstractModule implements ServiceGuiceSupport
{
    private final Environment environment;

    public TourOfHeroesModule(Environment environment) {
        this.environment = environment;
    }

    @Override
    protected void configure() {
        if (environment.isProd()) {
            bind(ServiceLocator.class).to(AkkaDiscoveryServiceLocator.class);
        }
    }
}
```

In the development mode, Lagom provides a special service locator. However, it requires adding

build.sbt

```
val lagomSvcRegistry = "com.lightbend.lagom" %% "lagom-service-registry-client" %
"1.6.1"
```

explicitly to the project's dependencies:

build.sbt

```
lazy val `toh-lagom-java-impl` = (project in file("toh-lagom-java-impl"))
  .enablePlugins(LagomScala, EcrPlugin)
  .dependsOn(`toh-lagom-java-api`)
  .configs(IntegrationTest)
  .settings(
    libraryDependencies ++= Seq(
      lagomJavadslServer,
      lagomSvcRegistry % Runtime,
      lagomJavadslPersistenceCassandra,
      lagomJavadslPersistenceJdbc,
      lagomJavadslAkkaDiscovery,
      kubernetesApi % Runtime,
      postgresql % Runtime
    )
  )
  .settings(
    Defaults.itSettings,
    TestSettings.forked(IntegrationTest),
    testOptions in Test += Tests.Argument(TestFrameworks.JUnit, "-q", "-v"),
```

```
    testOptions in IntegrationTest += Tests.Argument(TestFrameworks.JUnit, "-q", "-
v"),
    libraryDependencies ++= Seq(
      junit % Test,
      novocode % Test,
      novocode % IntegrationTest,
      lagomTestKit % IntegrationTest,
      h2 % IntegrationTest
    )
  )
  .settings(
    dockerBaseImage := "adoptopenjdk/openjdk11",
    packageName in Docker := (name in ThisBuild).value,
    dockerExposedPorts in Docker := Seq(9000, 9008, 8558, 2552, 25520),
    mappings in Universal += file("cassandra_truststore.jks") ->
"cassandra_truststore.jks",
    javaOptions in Universal ++= Seq(
      "-Dpidfile.path=/dev/null",
      "-Dconfig.resource=production.conf",
      s"-Dplay.http.secret.key=${Random.alphanumeric.take(40).mkString}"
    )
  )
  .settings(
    region in Ecr := AmazonUtils.awsRegion,
    repositoryTags in Ecr ++= Seq(version.value),
    repositoryName in Ecr := (packageName in Docker).value,
    login in Ecr := ((login in Ecr) dependsOn (createRepository in Ecr)).value,
    push in Ecr := ((push in Ecr) dependsOn (publishLocal in Docker, login in Ecr
)).value,
    localDockerImage in Ecr := s"${(packageName in Docker).value}:${(version in
Docker).value}"
  )
```

Oddly enough, Lagom's Java test kit library insists on having an in-memory H2 JDBC driver available as a runtime dependency, even when running in the Cassandra-only mode. Hence the inclusion of `val h2 = "com.h2database" % "h2" % "1.4.200"` as an `IntegrationTest` dependency above.

This snippet from `build.sbt` also shows some changes that are necessary to use JUnit for testing. This will be covered in the next chapter.

42. Testing With JUnit

Instead of ScalaTest, we use JUnit for testing in the Java version. We add the following dependencies in order to use JUnit in an SBT project:

build.sbt

```
val novocode = "com.novocode" % "junit-interface" % "0.11"
val junit = "junit" % "junit" % "4.13"
```

The JUnit library itself is there to make the JUnit code compile, of course. But in order for the JUnit tests not to be skipped by the `sbt test` and `sbt it:test` tasks, the Novocode library is necessary.

Besides the libraries, `TestFrameworks.JUnit` must be added to `testOptions` in both `Test` and `IntegrationTest` scopes for the `toh-lagom-java-impl` subproject.

build.sbt

```
  .settings(
    Defaults.itSettings,
    TestSettings.forked(IntegrationTest),
    testOptions in Test += Tests.Argument(TestFrameworks.JUnit, "-q", "-v"),
    testOptions in IntegrationTest += Tests.Argument(TestFrameworks.JUnit, "-q", "-
v"),
    libraryDependencies ++= Seq(
      junit % Test,
      novocode % Test,
      novocode % IntegrationTest,
      lagomTestKit % IntegrationTest,
      h2 % IntegrationTest
    )
  )
```

Once that is done, we can write JUnit tests, both unit and integration, as usual. Here is, for example, how the counterpart of the Scala integration test suite looks like. Note the use of Java functional futures.

toh-lagom-java-impl/src/it/java/com/chariotsolutions/tohlagom/impl/TourOfHeroesServiceTest.java

```
package com.chariotsolutions.tohlagom.impl;

import org.junit.Test;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import java.util.stream.Collectors;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.ExecutionException;

import com.chariotsolutions.tohlagom.api.Hero;
```

```
import com.chariotsolutions.tohlagom.api.NewHero;
import com.chariotsolutions.tohlagom.api.TourOfHeroesService;
import static com.chariotsolutions.tohlagom.common.Helpers.*;
import com.lightbend.lagom.javadsl.testkit.ServiceTest.TestServer;
import static com.lightbend.lagom.javadsl.testkit.ServiceTest.defaultSetup;
import static com.lightbend.lagom.javadsl.testkit.ServiceTest.startServer;
import static java.util.concurrent.TimeUnit.SECONDS;
import static org.junit.Assert.*;

public class TourOfHeroesServiceTest {
    private static TestServer server;
    private static TourOfHeroesService service;

    private final String name1 = "aLiCe";
    private final String name2 = "bOrIs";
    private final String name3 = "aNnA";
    private final String newName = "cOrInNe";
    private final String prefix = "A";

    @BeforeClass
    public static void setUp() {
        server = startServer(defaultSetup().withCassandra());
        service = server.client(TourOfHeroesService.class);
    }

    @AfterClass
    public static void tearDown() {
        if (server != null) {
            server.stop();
            server = null;
        }
    }

    @Test
    public void createHeroes() throws InterruptedException, ExecutionException,
TimeoutException {
        final Hero hero1 = service.createHero().invoke(new NewHero(name1))
                .toCompletableFuture().get(10, SECONDS);
        final Hero hero2 = service.createHero().invoke(new NewHero(name2))
                .toCompletableFuture().get(10, SECONDS);
        final Hero hero3 = service.createHero().invoke(new NewHero(name3))
                .toCompletableFuture().get(10, SECONDS);

        assertEquals(toTitleCase(name1), hero1.name);
        assertEquals(toTitleCase(name2), hero2.name);
        assertEquals(toTitleCase(name3), hero3.name);
    }

    @Test
    public void fetchHeroes() throws InterruptedException {
        Thread.sleep(30000);
```

```
        service.heroes().invoke().thenApply(heroes -> {
            assertEquals(3, heroes.size());
            return heroes.stream().map(hero -> hero.name).collect(Collectors.toSet());
        }).thenAccept(names -> {
            assertTrue(names.contains(toTitleCase(name1)));
            assertTrue(names.contains(toTitleCase(name2)));
            assertTrue(names.contains(toTitleCase(name3)));
        });
    }

    @Test
    public void searchHeroes() {
        service.search(prefix).invoke().thenAccept(heroes -> {
            assertEquals(2, heroes.size());
            heroes.forEach(hero -> assertEquals(toTitleCase(prefix), hero.name
.substring(0, prefix.length())));
        });
    }

    @Test
    public void fetchHero() {
        service.heroes().invoke().thenAccept(heroes -> {
            final String heroId = heroes.get(0).id;
            final String heroName = heroes.get(0).name;
            service.fetchHero(heroId).invoke().thenAccept(hero -> {
                assertEquals(heroId, hero.id);
                assertEquals(toTitleCase(heroName), hero.name);
            });
        });
    }

    @Test
    public void changeHero() {
        service.heroes().invoke().thenAccept(heroes -> {
            final String heroId = heroes.get(0).id;
            service.changeHero().invoke(new Hero(heroId, newName)).thenAccept(__ -> {
                try {
                    Thread.sleep(30000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                service.fetchHero(heroId).invoke().thenAccept(hero -> {
                    assertEquals(heroId, hero.id);
                    assertEquals(toTitleCase(newName), hero.name);
                });
            });
        });
    }

    @Test
```

```
    public void deleteHero() {
        service.heroes().invoke().thenAccept(heroes -> {
            final String heroId = heroes.get(0).id;
            service.deleteHero(heroId).invoke().thenAccept(__ -> {
                try {
                    Thread.sleep(30000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                service.heroes().invoke().thenAccept(remaining -> {
                    assertEquals(2, remaining.size());
                    assertFalse(remaining.stream().anyMatch(hero -> hero.id.equals
(heroId)));
                });
            });
        });
    }
}
```