Building a light management system with Scala + Akka + Finatra

Proof-of-concept implementation

version 1, 2016-08-16 by Wojciech Klaudiusz Zaborowski

Table of Contents

Introduction

What and why

As part of Philips Lighting recruitment process, I was asked to explore technology stack composed of:

- Scala (programming language): http://scala-lang.org/
- Akka (concurrent programming framework): http://akka.io/
- Finatra (webservices framework created at Twitter): http://twitter.github.io/finatra/

The most basic goal of the exercise was to find out how Finatra can be used to publish Akka-based application as webservices. It was my feeling however that to witness all nuts-and-bolts of Finatra-Akka integration one would need some non-trivial business logic implemented in Akka first. For keeping this experiment close to real life (and especially close to real life in Philips Lighting) I have spent some time familiarizing myself with Philips Hue system, which is an Internet-of-Things approach to light management:

http://www.developers.meethue.com/philips-hue-api

Then I build some pretty simple Akka model loosely inspired by Philips Hue. On top of that I placed Finatra and I spent some time crafting the webservice.

This document describes the resulting implementation and delves into various issues I solved on my way to the final success.

For keeping this document self contained, here is the technological context:

- Scala programming language: http://scala-lang.org/
- Akka (actor-based concurrency framework for Scala): http://akka.io/
- Finatra (webservices framework built by Twitter): http://twitter.github.io/finatra/

Intended audience

This document was written for software developers. Assumed is familiarity with: Scala, Akka, HTTP protocol and REST webservices.

The whole thing zipped to a 10 minutes tutorial

If you are one of these readers desperately wanting an extremely fast snapshot of the crucial knowledge, postponing reading of the whole (pretty long) document for next Saturday evening, here is the "10-minutes-tutorial" (be aware that this is SEVERE simplification of the whole story).

If you want rather detailed, slow introduction and everything properly explained, I recommend just skipping this chapter !

Summary of my achievements in this projects:

- I created a pattern for Finatra-Akka integration which may be used for any project. The pattern is not biased by business logic of my example application.
- Using this pattern I implemented a simple light management system (inpired by Philips Hue).
- This system exposes two APIs: one is purely Akka-based, another is just a REST webservice.
- I showed how to join DDD (domain-driven-design) and internet-of-things using Akka actors.
- I implemented pub-sub framework for Akka that follows DDD and Smalltalk style (as opposed to event bus style you can find in Akka itself).
- I designed a notation for Akka actors design that nicely integrates with UML class diagrams.

Starting the webservice:

- Clone this GIT repository: https://github.com/selfdual-brain/smart-light
- Start SBT inside the project main directory
- Use "run" command inside SBT shell to start the application
- Application startup will end up by publishing a REST webservice on localhost (port 8888)

Playing with the webservice:

- I prepared a collection of sample requests using the popular webservice testing tool Postman (https://www.getpostman.com/). This collection is stored inside Github repo, look at doc/webservice-sample-requests.json
- Class **com.selfdualbrain.finatrapoc.SmartLightWebserviceIntegrationTest** implements some unit tests for the webservice; you will find some request templates there
- You may run the unit tests from your IDE (or from SBT) – I assume you know how to do it

Browsing the source code:

- Akka actors are in **package com.selfdualbrain.finatrapoc.core.actors**
- The layer that pretends talking to actual Philips Hue bridges is implemented here: **`com.selfdualbrain.finatrapoc.core.hal`**
- "Finatra" layer consists of 3 classes:
  - entry point: **com.selfdualbrain.finatrapoc.webservice.SmartLightServer**
  - Guice config: **com.selfdualbrain.finatrapoc.core.EngineModule**
  - definition of the webservice: **com.selfdualbrain.finatrapoc.webservice.SmartLightController**
- Check the beginning of **SmartLightController** class – there you will find nice table explaining the mapping between webservice layer and Akka layer

Cherry on the pie:

- Two crucial methods, where the whole Finatra-Akka integration happens (this was the central goal of the whole experiment !):
  - **SmartLightController.akkaProcessing_Ask**
  - **SmartLightController.akkaProcessing_Tell**
- The only non-trivial thing here is that Finatra uses own implementation of Futures (different from Scala futures).
  - This class does the trick of futures conversion: **com.selfdualbrain.finatrapoc.utils.TwitterFutureConverters**

Stating the problem

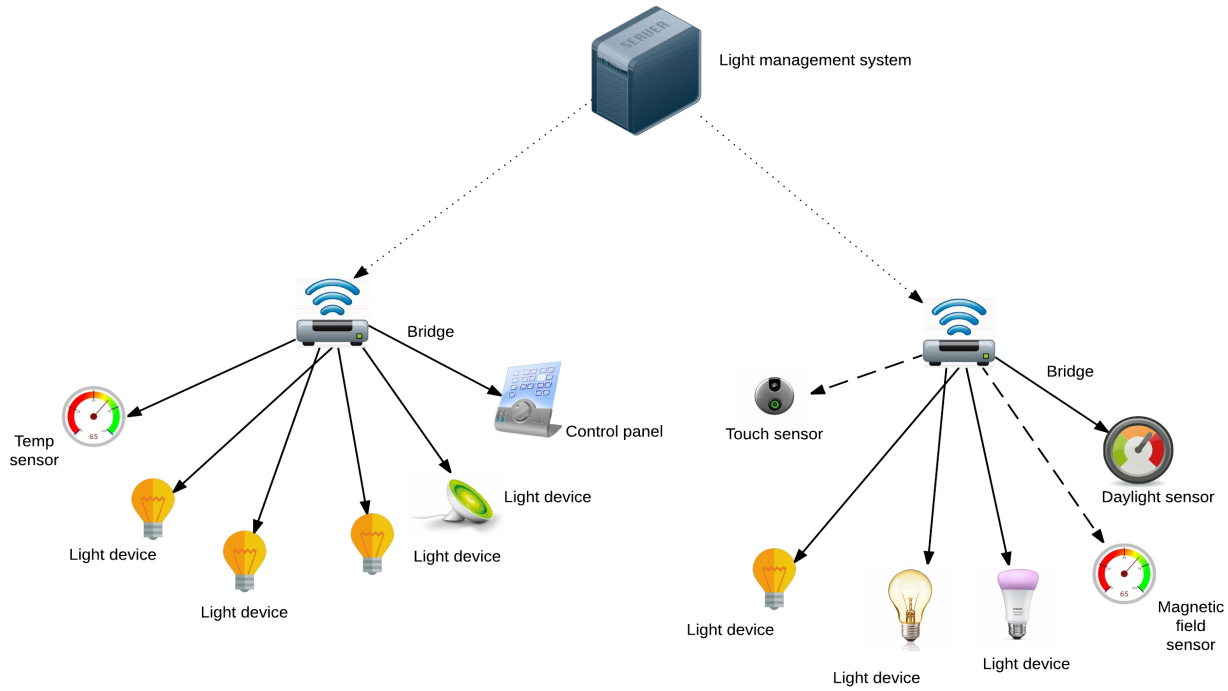
Motivation

Let me first explain the "imaginary problem" I wanted to build my implementation around.

Philips Hue is a system of smart lighting. The system is composed of various hardware devices: lights, sensors, user interface devices and bridges.

Bridges work like communication hubs. They are able to communicate with other devices (lights, sensors and user interface) using custom, low-level communication protocol. On the other hand each bridge exposes an API to the outside world (usually a webservice). Bridges however are unable to communicate with each other.

If you have a lighting system composed of several bridges, then you obviously come with a need to have a higher layer of logic that will allow you to manage your system as a whole. This was the basic idea behind this proof-of-concept solution I took as a base for testing Akka and Finatra technology stack.



So the idea was: we have a (possibly huge) collection of bridges, and every bridge managing up to 50 devices (lights, sensors, user interface pads). We want to expose a uniform interface to the lighting system composed of all these bridges. The system should expose the functionality as a webservice, so on top of this layer a system-wide lighting policies and algorithms may be operating. The system should be smart about handling diversity of devices, so that common functionality is accessible in an uniform way, but at the same time the diversity of features offered by different devices is not compromised.

The system should be dynamic in nature (= hardware components may be added and removed “on the fly”) and flexible (= able to handle diversity of types of hardware and hardware interfaces).

And specifically I wanted to achieve all this with the assumed technology stack: Scala + Akka + Finatra.

Big questions

Main questions to answer here were:

- How Akka fits into such a system ? It is fitting at all ? What patterns of using Akka are applicable ?
- How to integrate Finatra and Akka ?

To get a hint that above questions are quite non-trivial, one should keep in mind that:

- Akka was NOT created for hardware management in mind. Akka is just some general-purpose concurrent processing framework based on actor model.
- Although the actor model behind Akka is pretty old (developed in 1986 in the context of Erlang and Ericsson), Akka itself is relatively new tool and patterns of using Akka are not yet well established.
- Finatra was created in Twitter a long time before Akka was released.
- Finatra is using its own implementation of Future and Try, because at the time of its creations Scala was not having these yet (!) So they run into conflict with currently existing native Scala implementations of Future and Try.
- Typesafe (a company behind Scala and Akka) if actively developing its own solution for webservices development – previously named “Spray”, now rewritten under a new title “Akka-HTTP”). Going for Finatra+Akka mixture instead is somewhat against the “mainstream”, not a common pattern, so this is slightly like discovering America again.

Vision of the system

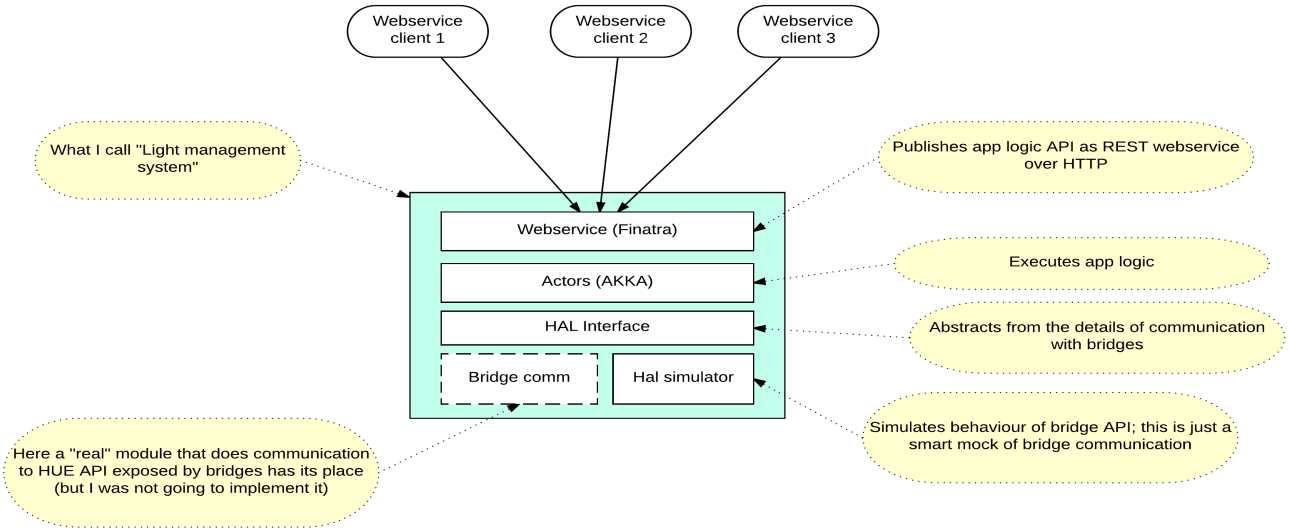
Given that I wanted just a proof-of-concept implementation, I faced the first obstacle:

- I wanted to limit myself to quite limited subset of features and requirement, so that anyone

analyzing my solution could read it and understand in reasonably short time
- I needed a model complex enough to discover hopefully all problems one would face in a production-scale solution

What I wanted to build

This picture summarizes my state of mind before actually starting the development effort:



Devices and features

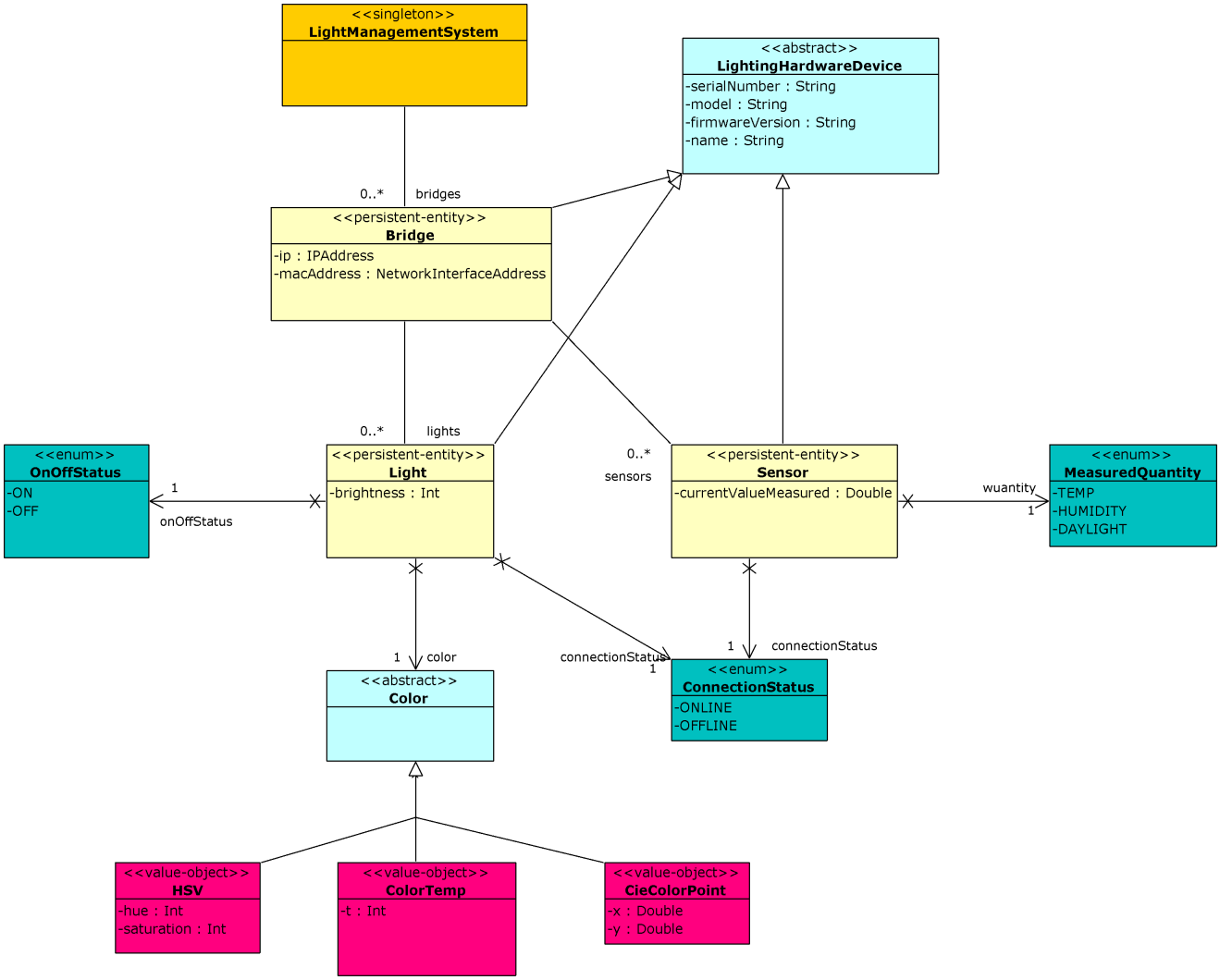
I limited the diversity of devices to be reflected in my proof-of-concept to:
- lights
- sensors
- bridges

I assumed that devices have some basic identification information:
- name
- serial id
- model name
- firmware version

I assumed the following features:
- lights can be turned on and off
- for a light one can set brightness and color
- I followed color spaces complexity as defined in HUE system API
- sensors can measure different quantities (like temperature or light intensity or others)
- bridges stand as front-end to lights and sensors
- Light/sensor can be connected to only one bridge
- Connection problems happen

Domain model

I came up with the following initial domain model (still conceptual):



Above model had to be however translated to implementation model which:

- introduce explicit Hardware Abstraction Layer
- use Akka actors
- expose API that can be consumed by the webservice layer

The most heavy question here was of course the Akka part. Akka gives a lot of freedom which means a lot of possibilities...

My approach to using Akka

Choosing the principal design pattern

Obvious choices were:

1. **Actors as virtual processors** – this is very natural when you think of actors as a virtualization of processing power. This is the most straightforward approach when you want to use Akka primarily to build a custom grid solution (with many computers). Requests correspond to messages but actors do not contain business logic, they only execute "workers" layer (= where the business logic is placed). This way there is a clear separation of business logic and actors layer – business logic never interferes with Akka.
2. **Actors pool per business facade** – this is actually Akka-based replica of "session-beans" pattern known from Java EE. There are "business logic" facades and for each facade there is a pool of actors which handle incoming requests. To make it under Akka you need to create one actor class for the facade itself (or preferably – use built-in so called Akka routers) and one worker class (these will sit in pools).
3. **Actor per request** – this is very natural, when you think of actors as "lightweight threads". You

create a new actor for each incoming webservice request. This actor - let's call it “request handler actor” - manages the whole complexity of business logic execution for one type of request. You end up having as many actor classes as types of requests. You can then easily share processing logic by actors class inheritance.

4. **Actor per domain object** – this is very natural when you follow “object oriented” or “domain driven development” way of thinking. So actors represent “live resources” and keep the state, whereas requests correspond to Akka messages. You end up mapping domain classes 1-to-1 to actor classes.

Approach (1) is in theory very interesting, because it keeps business code isolated from actors layer. Sounds appealing, right ? Yeah, but there is also a high price to be paid here – you are either losing the power of doing things asynchronously (when your workers are implemented in “classic way”) or you deal with a lot of futures in your business code. Actually I have seen some interesting examples of this approach in practice, but one may reasonably argue that this is definitely NOT what Akka creators had in mind when they were implementing Akka.

Approach (2) is in fact a variant (3) with some performance optimization applied (makes sense if request handler actor creation is expensive).

Approach (3) is probably most popular approach across the industry. Because is scales well during development as well as runtime and it is so much stateless-and-no-data-sharing in spirit. And all your actors are short living creatures – what a relief, right ? Well, however this approach has some … issue. Which may be no issue at all but you at least have to be aware of the problem: synchronization is pushed to the lower layer. Look – this is simple – every request spawns a new request-handler-actor right ? So all these actors try to execute their job in parallel. In effect we have a huge race condition situation across the app. And handling of this is completely left to another layers – to the database or distributed transaction monitor or to some simple optimistic locking logic done by some ORM or something like this. Which – again - may be completely okey if you are implementing a database-only application (99% of internet apps – including Whatsapp and Facebook fall into this group). Problems start if you have also some external resources other than just your master database. Like – hardware devices (for example). Or external banking system. Then you have to augment your clean actor-per-request approach with some smart synchronization tricks and you have to actually implement these tricks in your actors layer. And this may be tricky.

Approach (4) is of course something that first of all looks like fun. Instead of using Akka as a source of virtual processing power or as some lower layer, we actually drop classic approach to programming and instead we program business logic directly inside actors, using actors as “live objects” and Akka message send as primary way of components cooperation (instead of method calls). A pure domain driven, object oriented approach, Smalltalk-style in spirit. Is this possible at all ? What types of problems one encounters when following this approach ? Obviously this was an appealing adventure to try going this way, given that I was going to build a proof of concept, and the whole goal is about technology hacking experiment. But one has to admit that mapping bridges, sensors and lights to actor instances has one immediate nice effect – we eliminate race conditions around calling hardware. A light source is just a device – this is not a database. You can't rollback a “please switch on the light” operation. So achieving a full serializability of actions per device is a nice side effect of domain-driven approach and we are getting it for free.

My final decision was (4). It looked less obvious than (1)..(3) and more fun. And if this was going to suck in practice, I thought my experiment would potentially throw some light on this “suck” aspect.

Mapping internet-of-things to Akka

In the world of smart lighting systems, which itself is a perfect materialization of Internet-of-things concept:

https://en.wikipedia.org/wiki/Internet_of_things

… it should not be a surprise that for designing my actors layer I envisioned the following principle, which I will later refere to as “akka-of-things” (for the lack of a better name). Stated explicitly:

In the world of hardware devices we want each single device to be mapped to an actor. The implementation should give the impression that actors are devices. Send a message to an actor when you want a device to do something and expect a device sending messages when it wants to announce something. An actor should encapsulate the communication with the underlying device in such a way

that device-actors layer makes a functionally complete API for talking to devices.

The emerging beauty of this solution comes apparent when you realize that – for a system following akka-of-things principle – you can implement device management and generally any complex behavior of a system composed of (millions of) devices using purely actors sending messages.

Ok, now let's see how this is going to look in practice. Let's assume we have a **LightActor** class and a single instance of this class corresponds to a single Philips Hue light device. Now we are going to design the API of this device-actor.

incoming message	possible reply messages	what it does ?
GetDeviceInfo	DeviceInfo(serialNumber: String, name: String, model: String, isConnected: Boolean, brighness: Int, color: Color, effect: String, onOffStatus: Boolean)	retrieves complete information about this device
AdjustLight(brighness: Int, color: Color, effect: String)	DeviceInfo(name: String, model: String, isConnected: Boolean, brighness: Int, color: Color, effect: String, onOffStatus: Boolean) EffectNotSupported(name: String) BrightnessOutsideSupportedRange(value: Int, supportedRangeMin: Int, supportedRangeMax: Int)	adjusts the and brightness color of the light
SwitchOn	(NO MESSAGES)	switches on the light using the last color and brighness settings
SwitchOff	(NO MESSAGES)	switches off the light

This API looks great … but … something is missing here ! This something becomes more apparent when we want to build the API for a sensor device using the same “style”:

incoming message	possible reply messages	what it does ?
GetDeviceInfo	DeviceInfo(serialNumber: String, name: String, model: String, isConnected: Boolean, measuredQuantity: Quantity, currentValue: Double)	retrieves complete information about this device

Still works but to be able to read the temp value of a temp sensor we have to ask the sensor. And what if the temp is the same for hours ? We are asking and asking and asking … flooding Akka with polling messages. Which is definitely not needed here.

What we miss here is the events broadcasting mechanism. Something which will announce to the world “hey, I just measured a change in temperature” and others can selectively subscribe/unsubscribe to announcments. So this is a well known pub-sub pattern, the term “events broadcasting” is also used.

Interestingly, Akka comes with some built-in pub-sub solution:

http://doc.akka.io/docs/akka/2.3.15/scala/event-bus.html

Making the longs story short, I gave up using the the build-in Akka event buses solution for two reasons:

- the way they built it was not precisely what I think we need; I wanted each actor-device to be able to throw events, whereas Akka event buses are a separate construct
- I wanted to keep this simple for this POC to stay easy to digest; using separate Akka event bus for each actor looked as an overkill

Fortunately, it is quite easy to implement a suitable event broadcasting feature with Akka.

Implementing event system for Akka

I wanted a very simple events broadcasting solution:

- when an actor announces that something happened, it broadcasts, or “triggers” a message (the latter term follows Smalltalk naming)
- any actor A can subscribe for receiving messages broadcasted by actor B:
  - A sends message Subscribe to B to start subscription

    - A sends message Unsubscribe to B to cancel subscription
- there is no events filtering (to keep things simple) – if A subscribed to B, it will be getting all events triggered by B
- an actor may decide to re-broadcast all events from another actor (imagine an engine actor and car actor and car re-broadcasts messages from engine)
- each event knows the origin, i.e. the actor that originally triggered this event (re-broadcasting is not changing the origin)

The whole thing is implemented in trait **com.selfdualbrain.finatrapoc.utils.EventsBroadcaster**. If a developer wants to use events broadcasting in a given actor class, all he has to do is just mixing-in this trait to the actor class.

Let's see an example of events broadcasting in action:

```
15  class LightSystemManagerActor(bridgeDiscovery: BridgeDiscoveryApi) extends Actor with ActorLogging with EventsBroadcaster {
16    var bridges: mutable.Map[DeviceUniqueId, ActorRef] = new mutable.HashMap[DeviceUniqueId, ActorRef]
17
18    for (deviceInfo <- bridgeDiscovery.discoverAllBridgesInScope)
19      createBridgeActor(deviceInfo, bridgeDiscovery.getBridgeInterface(deviceInfo.serialNumber))
20
21    override def receive: Receive = super.receive orElse {
22      case Req.ListConnectedBridges =>
23        log.debug("light system manager: got ListConnectedBridges request")
24        sender ! Res.ConnectedBridges(bridges.toMap)
25    }
26
27    override protected def handleEvent(origin: ActorRef, payload: Any): Unit = {
28      log.debug(s"EVENT triggered by ${origin.path.name}: $payload")
29    }
30
31    def createBridgeActor(deviceInfo: BridgeInfo, hardwareApi: BridgeHardwareApi): Unit = {
32      val newChildActor = context.actorOf(BridgeActor.props(hardwareApi, deviceInfo), "bridge-" + deviceInfo.serialNumber.value)
33      bridges += deviceInfo.serialNumber -> newChildActor
34      this.listenAndReBroadcastEventsFrom(newChildActor)
35      trigger(LightSystemEvent.NewBridgeConnected(deviceInfo.serialNumber))
36    }
37  }
38
39  object LightSystemManagementActor {
40    def props(bridgeDiscovery: BridgeDiscoveryApi) = Props(new LightSystemManagerActor(bridgeDiscovery))
41
42    object Req {
43      case object ListConnectedBridges
44    }
45
46    object Res {
47      case class ConnectedBridges(map: Map[DeviceUniqueId, ActorRef])
48    }
49
50  }
51
```

Line 15: Notice the “with EventsBroadcaster” part – we are making this actor class capable of events broadcasting.
Line 35: Triggering of new event.
Line 34: Subscribing in “listen and broadcast” mode.
Line 27: Handling events.

For details go to scaladoc in EventsBroadcaster trait.

Design notation for Akka

For years UML notation was considered quite useful and it seems like still being around … but the the context of Akka the community and especially Typesafe looks like not having a clear preference of design notation.

Personally, I still find class diagrams very useful in the context of Akka. You just have to introduce a couple of simple conventions that I am describing here.

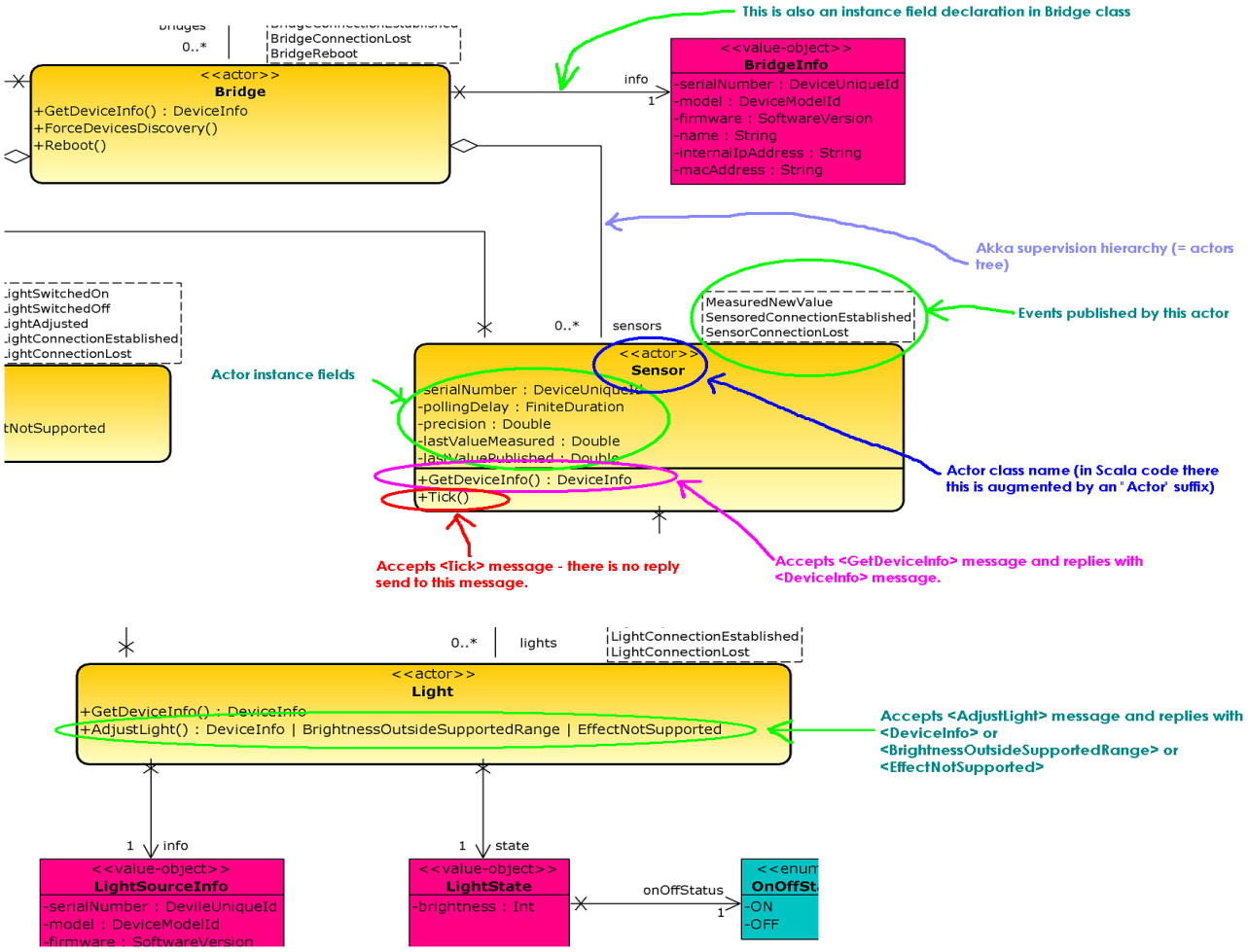
Starting from the basics – an actor class is just a class, so it should naturally fit into a class diagram. We can use stereotypes (plus some graphical appearance adjustments) to make actor classes easy to distinguish from “normal” classes. For this I use “actor” stereotype, an orange color and rounded corners of class “box”.

For “normal” UML classes we have attributes and methods. For actors we need:

- attributes
- accepted and replied messages

- broadcasted events
- supervision hierarchy (= akka actors hierarchy)

Look how I achieved all this using pretty mainstream UML editor (it is Visual Paradigm for UML, but pretty any UML editor should do the job):



Coding conventions I used for actors layer

```
15  class BridgeActor(val bridgeHardware: BridgeHardwareApi, val info: BridgeInfo) extends Actor with ActorLogging with EventsBroadcaster {
16    val lights: mutable.Map[DeviceUniqueId, ActorRef] = new mutable.HashMap[DeviceUniqueId, ActorRef]
17    val sensors: mutable.Map[DeviceUniqueId, ActorRef] = new mutable.HashMap[DeviceUniqueId, ActorRef]
18
19    //initalization of child actors
20    for (device <- bridgeHardware.getAllLights)
21      this.createLightActor(device.serialNumber)
22    for (device <- bridgeHardware.getAllSensors)
23      this.createSensorActor(device.serialNumber)
24
25    log.info(s"actor for bridge ${info.serialNumber.value} is ready, actor-path=${self.path}")
26
27    override def receive: Receive = super.receive orElse {
28      case Req.GetDeviceInfo =>
29        sender ! Res.DeviceInfo(info, lights.toMap, sensors.toMap) //sending immutable copies of lights and sensor maps
30
31      case Req.ForceDevicesDiscovery =>
32        for (newLightDevice <- bridgeHardware.searchForNewLights())
33          this.createLightActor(newLightDevice.serialNumber)
34
35      case Req.Restart =>
36        bridgeHardware.reboot()
37        trigger(LightSystemEvent.BridgeReboot)
38
39    }
40
41    def createLightActor(serialNumber: DeviceUniqueId) = {
42      val newChildActor = context.actorOf(LightActor.props(bridgeHardware, serialNumber), "light-" + serialNumber.value)
43      lights += serialNumber -> newChildActor
44      this.listenAndReBroadcastEventsFrom(newChildActor)
45      trigger(LightSystemEvent.NewLightDiscovered(serialNumber))
46    }
47
48    def createSensorActor(serialNumber: DeviceUniqueId) = {
49      val newChildActor = context.actorOf(SensorActor.props(bridgeHardware, serialNumber, Config.sensorsPollingInterval, Config.sensorsPrecision),
50      sensors += serialNumber -> newChildActor
51      this.listenAndReBroadcastEventsFrom(newChildActor)
52      trigger(LightSystemEvent.NewSensorDiscovered(serialNumber))
53    }
54
55  }
56
57  object BridgeActor {
58    def props(bridgeHardware: BridgeHardwareApi, info: BridgeInfo) = Props(new BridgeActor(bridgeHardware, info))
59
60    object Req {
61      case object GetDeviceInfo
62      case object ForceDevicesDiscovery
63      case object Restart
64    }
65
66    object Res {
67      case class DeviceInfo(info: BridgeInfo, lights: Map[DeviceUniqueId, ActorRef], sensors: Map[DeviceUniqueId, ActorRef])
68      case object Test
69    }
70
71  }
```

Line 15: I mix-in ActorLogging trait to achieve logging capability in actors.
Line 15: I mix-in EventsBroadcaster to use my events framework.
Line 57: I always define companion class for an actor class.
Line 58: I use the props convention as recommended by Akka docs (for actors creation)
Lines 60-69: I define actor message-based API in companion class:

- all messages ade declared as case classes
- I define 3 grouping singletons: **Req** for incoming messages (“request”), **Res** for responses, **Internal** for messages that are not part of public API

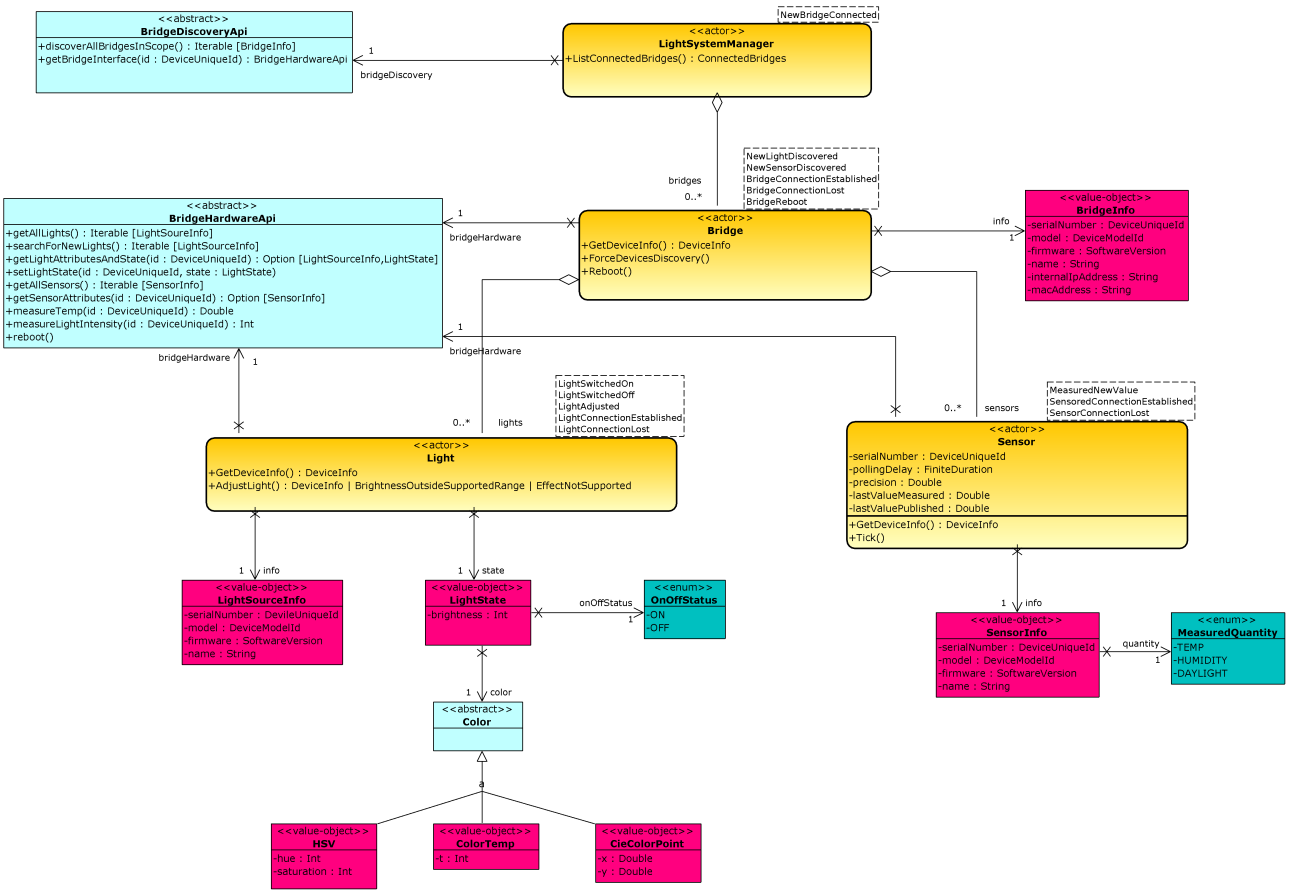
SmartLight actors model

Creating actors model for the POC, some decision were obvious: we wanted actors for bridges, lights and sensors.

Two things were less obvious, however.

First: I added a singleton LightSystemManager actor that sits stands like a “facade” for the whole lighting system and by the way it keeps the collection of all bridges.

Second: I was hesitating should bridge actor be a akka-supervisor actor for its sensors and lights or not ? Finally I opted for “yes” but this there is no clear winner here, one could equally well make all devices child actors of LightSystemManager.



Finatra approach to building webservices

This chapter contains a crash course on building webservices with Finatra. If you are familiar with Finatra, you can jump directly to the next chapter.

For making things work you of course need a collection of proper imports. Check the **build.sbt** if in doubts.

Step 1: Server class

Server class stands as an entry point for the application:

```
object SmartLightServerMain extends SmartLightServer

/**
  * Light system webservice server.
  * This is the entry point to the application.
  */
class SmartLightServer extends HttpServer {

  //  override val defaultFinatraHttpPort: String = ":8080"

  override val modules = Seq(EngineModule)

  override def configureHttp(router: HttpRouter) {
    router
      .filter[LoggingMDCFilter[Request, Response]]
      .filter[TraceIdMDCFilter[Request, Response]]
      .filter[CommonFilters]
      .add[SmartLightController]
  }


}
```

One must extend HttpServer class provided by Finatra. Two things must be defined here:

- Guice module (only this way we will be able to inject some stuff into controllers) – by overriding **modules** value.
- Collection of controllers (they define the webservice) – by overriding configureHttp() method. In my case I am just registering one controller – **SmartLightController**.

Step 2: Guice configuration

Finatra internally uses Google Guice for dependency injection and it is kinda forcing to to plug your code into existing injection infrastructure.

To be able to implement your controllers, you will need some components. And your Guice config is the place where these components should be created.

What you need to to is to create a standalone object that extends TwitterModule and then override **configure()** method.

```
object EngineModule extends TwitterModule {

  override def configure(): Unit = {
    val actorSystem = ActorSystem("galaxy")

    bind[ActorSystem].toInstance(actorSystem)

    val hal = SampleHardwareDefinition.halSimulator
    val rootActor = actorSystem.actorOf(LightSystemManagementActor.props(hal), "root-manager")

    bind[ActorRef].annotatedWith(Names.named("root-actor")).toInstance(rootActor)
  }

}
```

In above example I am bootstrapping the actorsystem which my actors run on.
I am also creating my hardware simulator and I am binding all these so to be able to inject these things into my controller.

Step 3: Controllers

Controllers is where your webservicee is actually implemented. For serious project is it likely that you will use controllers to modularize your webservice, but for my proof-of-concept I just created only one controller.

Look at the beginning of the controller class to see how I am doing dependency injection:

```
class SmartLightController @Inject() (actorSystem: ActorSystem, @Named("root-actor") rootActor: ActorRef) extends Controller {
```

The body of controller contains a collection of REST routes handlers. Well, they are pretty much like methods, just using a small DSL that Finatra defines for you. Example of such “method”:

```
  get("/bridge/:id") { request: Request =>
    debug(s"WEBSERVICE: ${request.method} ${request.uri}")

    val bridgeId = DeviceUniqueId(request.params("id"))

    val convertReplyMsg: BridgeActor.Res.DeviceInfo => Webservice.DeviceInfo = {msg =>
      Webservice.Bridge(
        uriPath = WsResourcePaths.bridge(msg.info.serialNumber),
        model = msg.info.model.value,
        name = msg.info.name,
        serialNumber = msg.info.serialNumber.value,
        firmwareVersion = msg.info.firmware.value,
        ip = msg.info.internalIpAddress,
        macAddress = msg.info.macAddress,
        lightsConnected = msg.lights.size,
        sensorsConnected = msg.sensors.size
      )
    }

    akkaProcessing_Ask(
      actorPath = ActorPaths.bridge(bridgeId),
      sendMsg = BridgeActor.Req.GetDeviceInfo,
      responseConverter = {
        case msg: BridgeActor.Res.DeviceInfo => response.ok(convertReplyMsg(msg))
      }
    )
  }
```

So this one is handling all requests with URI path matching the pattern **/bridge/NNNN**, and the NNNN parameter is captured so that I can use it in the body of my handler.

What is really cool is that Finatra does JSON parsing and rendering completely automatically, as long as you stick to their JSON mapping convention. So I did not have to create event a single line of code to deal with JSON format, actually (only exception to this are unit tests, where I obviously WANT to explicitly JSON). This is explained in more detail in next two steps.

Step 4: Consuming input when we ignore request body

This is the simplest case. So the input consists of:

- URI path parameters
- URI query parameters

Simple example explains it all. When your route handler starts with:

```
get(“/bridge/:a/light/:b”) { request: Request =>
   //some processing here
}
```

...then once a server is hit with a request:

```
GET http://localhost:8888/bridge/199932/light/2423423?info=full
```

all parameters passed as part of the URI are automatically captured by Finatra and merged into a single Map[String,String] that is available as **request.params**.

So you will get:

```
request.params(“a”) == “199932”
request.params(“b”) == “2423423”
request.params(“info”) == “full”
```

Step 5: Consuming input when we want to parse request body

In such case you just define the required structure of the body as a case class. There are 2 tricks here.

Trick 1 is really easy - where previosly we had **request: Request**, now we just change the declared type to our case class (which exists solely as a body structure definition):

```
190   post("/bridge/:bridge_id/light/:light_id/adjust") { request: Webservice.LightAdjustment =>
191     debug(s"WEBSERVICE: /bridge/${request.bridgeId}/light/${request.lightId}/adjust")
192
193     val bridgeId = DeviceUniqueId(request.bridgeId)
194     val lightId = DeviceUniqueId(request.lightId)
```

And look here how the request body definition was crafted:

```
64    case class LightAdjustment(
65      @RouteParam bridgeId: String,
66      @RouteParam lightId: String,
67      on: Boolean,
68      brightness: Int,
69      hue: Option[Int],
70      saturation: Option[Int],
71      xy: Option[(Double, Double)],
72      ct: Option[Int],
73      effect: String
74    )
```

Trick 2: is the way we capture route params. Previously (step 4) we had this nice **request.params** map, now the map is simply not there. There is however another approach here, also prepared by Finatra - using custom @RouteParam and @QueryParam annotations (as seen in the above example for @RouteParam).

Caution: Thanks to Finatra naming conventions there is however some small glitch here, which took me some debugging time: to make route params capture working this way, you have to use snake case naming convention inside the route pattern. Please observe that for **bridgeId** I am using camel-case inside the case class and snake-case inside the route pattern. Small detail but crucial !

Step 6: Rendering output

Finatra DSL also makes this very easy: there is a **response** keyword defined for you. What it really contains is a ResponseBuilder (whatever it is), which is finally very easy to use:

```
69    get("/bridge/:id") { request: Request =>
70      debug(s"WEBSERVICE: ${request.method} ${request.uri}")
71
72      val bridgeId = DeviceUniqueId(request.params("id"))
73
74      val convertReplyMsg: BridgeActor.Res.DeviceInfo => Webservice.DeviceInfo = {msg =>
75        Webservice.Bridge(
76          uriPath = WsResourcePaths.bridge(msg.info.serialNumber),
77          model = msg.info.model.value,
78          name = msg.info.name,
79          serialNumber = msg.info.serialNumber.value,
80          firmwareVersion = msg.info.firmware.value,
81          ip = msg.info.internalIpAddress,
82          macAddress = msg.info.macAddress,
83          lightsConnected = msg.lights.size,
84          sensorsConnected = msg.sensors.size
85        )
86      }
87
88      akkaProcessing_Ask(
89        actorPath = ActorPaths.bridge(bridgeId),
90        sendMsg = BridgeActor.Req.GetDeviceInfo,
91        responseConverter = {
92          case msg: BridgeActor.Res.DeviceInfo => response.ok(convertReplyMsg(msg))
93        }
94      )
95    }
```

Look at this fragment of line 92:

```
response.ok(convertReplyMsg(msg))
```

So the first thing here is that the response builder offers you a collection of HTTP return codes, provided as methods, so you have:

```
response.ok(...)
response.notFound(...)
response.internalServerError(...)
```

… and so on.

All these methods exist in two flavours - one that accepts a body (which must be a case class instance) and another with no body. JSON rendering - again - will happen on the fly, and Finatra is smart enough to properly handle Option[T], collections and aggregation (= nested objects).

Step 7: Writing unit tests

This is sample unit test for the server itself (extending **com.twitter.inject.server.FeatureTest** is the main trick here):

```
8   class SmartLightServerStartupTest extends FeatureTest {
9
10    override val server = new EmbeddedHttpServer(
11      twitterServer = new SmartLightServer,
12      stage = Stage.PRODUCTION,
13      verbose = false)
14
15    "Server" should {
16      "startup" in {
17        server.assertHealthy()
18      }
19    }
20  }
```

This is a sample unit test for the webservice:

```
8   class SmartLightWebserviceIntegrationTest extends FeatureTest {
9
10    override val server = new EmbeddedHttpServer(new SmartLightServer)
11
12    "Server" should {
13
14      "List connected bridges" in {
15        server.httpGet(
16          path = "/system/bridges",
17          andExpect = Status.Ok)
18      }
19
20      "Return bridge B-001 info" in {
21        server.httpGet(
22          path = "/bridge/B-001",
23          andExpect = Status.Ok)
24      }
25
26      "Return bridge B-002 info" in {
27        server.httpGet(
28          path = "/bridge/B-002",
29          andExpect = Status.Ok)
30      }
31
32      "Refuse getting info on non-existing bridge B-003" in {
33        server.httpGet(
34          path = "/bridge/B-003",
35          andExpect = Status.NotFound)
36      }
37
38      "List devices connected to bridge B-001" in {
39        server.httpGet(
40          path = "/bridge/B-001/devices",
41          andExpect = Status.Ok)
42      }
```

Step 8: Running the webservice

- Start SBT inside the project main directory
- Use “run” command inside SBT shell to start the application

Building SmartLight webservice

Webservice API vs Akka API

By now it should be clear that – following akka-of-things principle – in my solution the primary API for talking to devices is by sending messages to actors.

It would be a complete misunderstanding if one reads my experiment as “webservices building tutorial”. It is not. My line of thinking was from the beginning the opposite:

- establish akka-of-things as the guiding principle
- consider different mappings of akka-of-things API to other communication protocols (like Kafka, REST webservices, SOAP webservices, websockets, ESB etc.)

Now, regarding the Finatra integration part - once I established akka-of-things API for my Smart Light system, I wanted to look at the very simplest mapping of this API to REST webservice using Finatra.

My motivations explained shortly:

- I didn't want to create a functionally complete mapping of actors layer to webservice layer. I just picked up a subset of functionality to have a reasonably diverse set of example requests for the proof-of-concept
- mapping of events is not supported (for doing so within the webservice layer one would have to use some notifications framework or websockets – this is a longer issue, beyond the scope of this POC)
- I was specifically interested in recreating a real-life characteristics of the webservice ↔ actors border. And the real life is that both layers would evolve over time, and this evolution cannot go precisely in parallel – usually for backwards compatibility reasons. I wanted to see how my Finatra-Akka bridge will handle some non-trivial mapping issues - at least to see the top of the iceberg of problems that can emerge. For achieving this I sketched the webservice API actually before building the akka-of-things layer. For webservice layer I borrowed some fields and types from Philips Hue API, for akka-of-things I designed stuff more closely to Scala standards. I also added some “not supported” corner cases or obsolete fields, sort of artificial mess here and there.

Webservice API layout

These were REST requests patterns I wanted to implement:

Command pattern	Semantics
resource: SYSTEM	
GET /system/bridges	retrieves collection of connected bridges
resource: bridge/NNNN	
GET /bridge/NNNN	retrieves device info for this bridge
GET /bridge/NNNN/devices	lists devices (lights and sensors) connected to this bridge
POST /bridge/NNNN/reboot	reboots the bridge
resource: Light/XXXX	
GET /bridge/NNNN/light/XXXX	retrieves device info for this light
POST /bridge/NNNN/light/XXXX/adjust	adjust color and brightness of this light
POST /bridge/NNNN/light/XXXX/switch-on	turns on the light
POST /bridge/NNNN/light/XXXX/switch-off	turns off the light
resource: sensor/XXXX	
GET bridge/NNNN/sensor/XXXX	retrieves device info for this sensor

I followed the Finatra documentation (as explained in “Finatra approach to building webservices” chapter) and some design/implementation problems I encountered are worth describing – I devote next 4 sub-chapters for them.

Problem 1: Mapping between resources and actors

In webservice layer we have resources, whereas in actors layer we have actors. Thanks to akka-of-things approach these two could go more or less in 1-to-1 correspondence. And I knew that I will be needing conversion in both directions:

- when handling an incoming webservice request I needed to discover an actor corresponding to the resource in question
- when building webservice responses I needed to build URIs of resoures, given actor-refs as input

So, on one side we have URI paths, like **/bridge/123456**.
On the other side we have actor refs, like **akka://galaxy/user/root-manager/bridge-123456**
BTW - “galaxy” is the name of actorsystem (galaxy is a huge collection of lights, looked like a good name in the context of large-scale and futuristic light management system).

Converting webservice paths to actor paths and vice versa was a nightmare and even worse – it was not compliant with “single point of change” rule. Finally I discovered another approach to the problem, that worked surprisingly well. The source-code is self-explanatory:

```
8  trait PathsMapper {
9    def rootManager: String
10   def bridge(serialNumber: DeviceUniqueId): String
11   def light(bridge: DeviceUniqueId, light: DeviceUniqueId): String
12   def sensor(bridge: DeviceUniqueId, sensor: DeviceUniqueId): String
13 }
14
15 object ActorPaths extends PathsMapper {
16   override def rootManager: String = "akka://galaxy/user/root-manager"
17   override def bridge(serialNumber: DeviceUniqueId): String = s"akka://galaxy/user/root-manager/bridge-${serialNumber.value}"
18   override def light(bridge: DeviceUniqueId, light: DeviceUniqueId): String = s"akka://galaxy/user/root-manager/bridge-${bridge.value
19   override def sensor(bridge: DeviceUniqueId, sensor: DeviceUniqueId): String = s"akka://galaxy/user/root-manager/bridge-${bridge.val
20 }
21
22 object WsResourcePaths extends PathsMapper {
23   override def rootManager: String = "/system"
24   override def bridge(serialNumber: DeviceUniqueId): String = s"/bridge/${serialNumber.value}"
25   override def light(bridge: DeviceUniqueId, light: DeviceUniqueId): String = s"bridge/${bridge.value}/light/${light.value}"
26   override def sensor(bridge: DeviceUniqueId, sensor: DeviceUniqueId): String = s"bridge/${bridge.value}/sensor/${sensor.value}"
27 }
28
```

Problem 2: The mess with futures

As stupid as it sounds, it is a fact – Finatra has its own implementation of Futures. Well, for historical reasons – when they were crafting Finagle and Finatra at Twitter, Scala was so fresh and young that Futures were not implemented yet. OK, but why they did not refactored-out the old futures now, when a proper scala implementation is in place ? I have no idea.

To understand the reason why we actually have to deal with Finatra futures, we need to delve slightly deeper into the route handlers. Well, in my Finatra tutorial – sub-chapter “Step 6: Rendering output” I did not tell the whole truth.

Finatra is from the beginning designed to handle huge amount of concurrent clients. For dealing with high traffic it is based on futures, so everything is asynchronous. For making this happen, the route handler should actually encapsulate the processing in a future:

```
145   post("/bridge/:id/reboot") { request: Request =>
146     val bridgeId = DeviceUniqueId(request.params("id"))
147     com.twitter.util.Future {response.ok(s"got bridge: $bridgeId")}
148   }
```

I will rewrite the same piece of code again, but using explicit types for the callback, to make things more apparent:

```
150   val callback: Request => com.twitter.util.Future[Response] = { request: Request =>
151     val bridgeId = DeviceUniqueId(request.params("id"))
152     com.twitter.util.Future {response.ok(s"got bridge: $bridgeId")}
153   }
154
155   post("/bridge/:id/reboot")(callback)
```

So, callback is just a function and what Finatra does under the hood is checking if the return type of callback is Future[Result] as expected. If this is not the case – Finatra tries to automatically encapsulate the result in a future – it is easy to guess that in general this can't be done properly – this is just a fallback behavior. But for doing things properly, it is a developer's responsibility to construct a proper future, encapsulating the fragment of code that is subject to asynchronous execution.

Now it is at leas clear why we need to use Finatra futures. OK, so maybe we just can stick to using ONLY Finatra futures and problem is solved - ?

Well, not really. Because Akka is using normal (i.e. “scala”) futures ! So to be able to merge asychnorous semantics of Finatra with asychnchronous semantics of sendin messages to actors, we definitely need an ability to convert scala futures to finatra futures.

Although tricky, this is actually not terribly complicated – look at my implementation in class **TwitterFutureConverters**. The crucial piece of code is here:

```
27  implicit def scalaToTwitterFuture[T](f: Future[T])(implicit ec: ExecutionContext): twitter.Future[T] = {
28    val promise = twitter.Promise[T]()
29    f.onComplete(promise update _)
30    promise
31  }
```

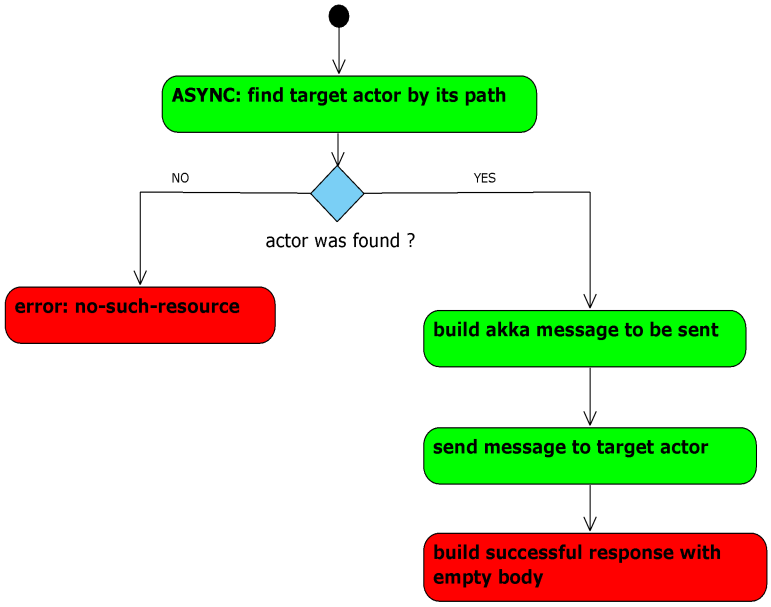
Problem 4: calling AKKA from webservice controller

This was the most tricky part (and in a sense the central point of the whole POC). I wanted to implement webservice handlers using lower-layer API – my akka-of-things of course.

Distinguishing ASK and TELL
Calling an actor has of course two flavours:

- ASK: send the message and wait for reply message
- TELL: send the message and forget (the actor will start processing but we are not going for the end of this processing)

Calling Akka with TELL semantics
This is the easier case. What we know is that we want to return and instance of Future[Result] (using Twitter version of the future). Let's do an execution plan of this piece of code that should be enveloped by the resulting future:



Of course some parts of this processing are going to repeat for every TELL situation, so we need some micro-DSL helping us to avoid code duplication.

It is also worth noticing that some parts of this processing are asynchronous (I marked one step with ASYNC). Which means that we CAN'T just implement above diagram as a sequential processing and encapsulate into a future - that would be a huge waste of Finatra power ! What we really need is using properly the algebra of futures.

Let's first look to an example of how a route handler using TELL semantics (and my micro DSL) effectively looks like:

```
133     post("/bridge/:id/reboot") { request: Request =>
134       debug(s"WEBSERVICE: ${request.method} ${request.uri}")
135
136       val bridgeId = DeviceUniqueId(request.params("id"))
137
138       akkaProcessing_Tell(
139         actorPath = ActorPaths.bridge(bridgeId),
140         sendMsg = BridgeActor.Req.Restart
141       )
142     }
```

And this is how **akkaProcessint_Tell** is implemented:

```
379   def akkaProcessing_Tell(actorPath: String, sendMsg: Any): com.twitter.util.Future[Response] = {
380     implicit val executionContext = actorSystem.dispatcher
381
382     val futureActorRef: Future[ActorRef] = actorSystem.actorSelection(actorPath).resolveOne(Config.finatraAkkaBridgePathResolvingTimeout)
383
384     val fut = (futureActorRef map {targetActorRef => targetActorRef ! sendMsg; response.ok}) recover {
385       case ex: ActorNotFound => response.notFound(Webservice.Error(
386         errorCode = "not such resource",
387         description = "the URI pointed to non-existing resource, this is client-side error",
388         diagnosticInfo = Some(s"request internally converted to akka actor path: $actorPath, but such actor was not found in the system")
389       ))
390     }
391
392     return TwitterFutureConverters.scalaToTwitterFuture(fut)(actorSystem.dispatcher)
393   }
```

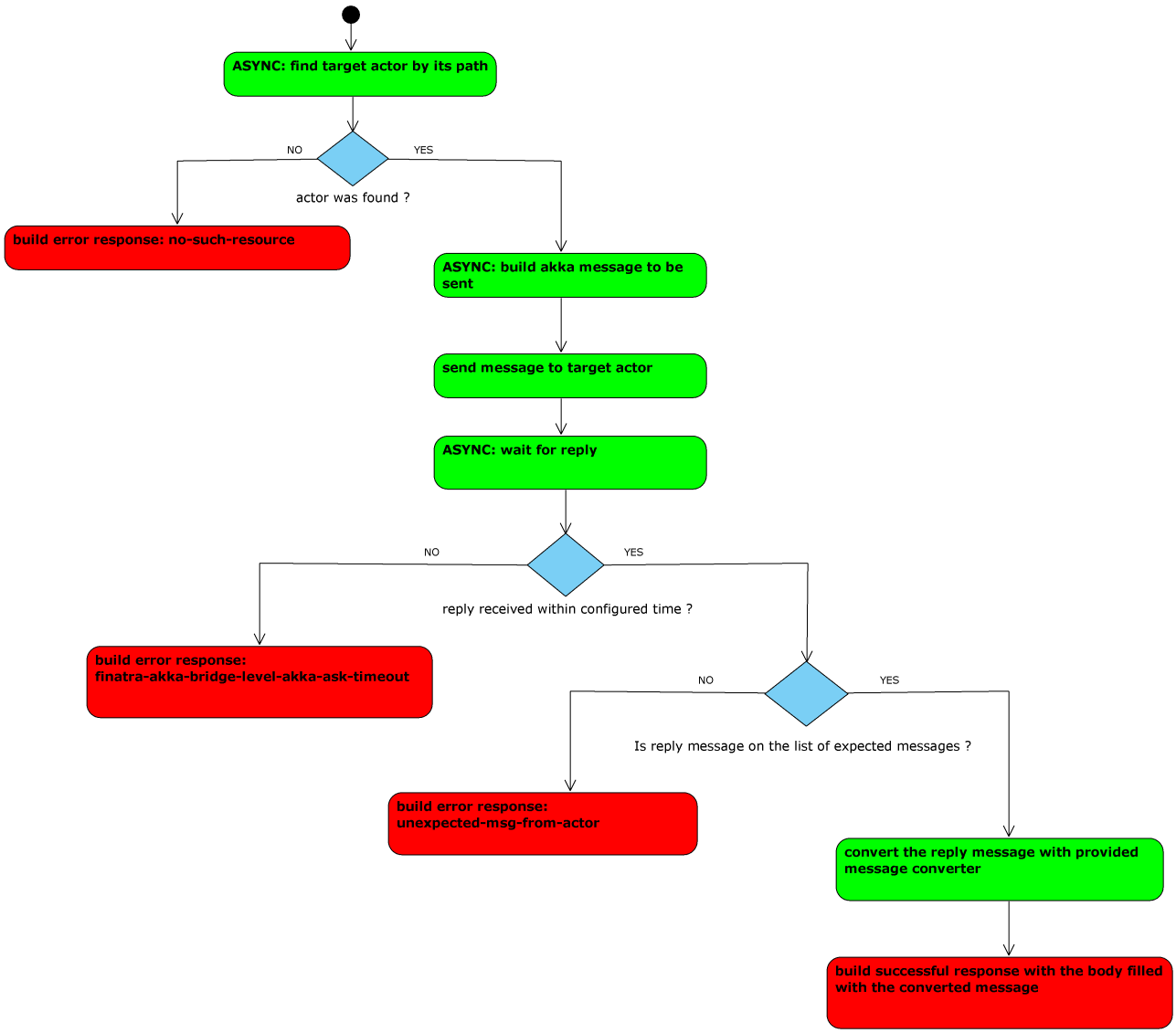
Calling Akka with ASK semantics
The same idea but more complications, because we have to wait for response from actor (in a non-blocking way !) and we have to handle all possible replies we will get.

Regarding replies, these are all possible cases we have to handle:

1. the reply never arrived (=timeout elapsed)
2. the reply arrived but it was not as we expected
3. the reply arrived and we have to understand:
    1. is this a “success” reply or
    2. is this a “failure” reply (actor gently refusing to do what we wanted, signalling application level “exception”)

It took me quite an intense research for finding a simple yet powerful syntax, but before I show the syntax

let's - as previosuly - try to understand what is the execution plan here:



Now look how I have all this happening with a very simple syntax sugar:

```
190   post("/bridge/:bridge_id/light/:light_id/adjust") { request: Webservice.LightAdjustment =>
191     debug(s"WEBSERVICE: /bridge/${request.bridgeId}/light/${request.lightId}/adjust")
192
193     val bridgeId = DeviceUniqueId(request.bridgeId)
194     val lightId = DeviceUniqueId(request.lightId)
195
196     val requestedColor: Color =
197       if (request.hue.isDefined)
198         HSV(request.hue.get, request.saturation.get)
199       else if (request.xy.isDefined)
200         CieColorPoint(request.xy.get._1, request.xy.get._2)
201       else if (request.ct.isDefined)
202         ColorTemp(request.ct.get)
203       else throw new Exception("color value is missing")
204
205     akkaProcessing_Ask(
206       actorPath = ActorPaths.light(bridgeId, lightId),
207       sendMsg = LightActor.Req.AdjustLight(brighness = request.brightness, color = requestedColor, effect = request.effect),
208       responseConverter = {
209         case msg: LightActor.Res.DeviceInfo => response.ok(convertLightDeviceInfo(bridgeId, msg))
210         case LightActor.Res.EffectNotSupported(effectName) =>
211           response.status(Status.UnprocessableEntity).body(Webservice.Error(
212             errorCode = "effect-not-supported",
213             description = s"effect $effectName is not supported for light device ${request.lightId}",
214             diagnosticInfo = None))
215         case LightActor.Res.BrightnessOutsideSupportedRange(value, supportedRangeMin, supportedRangeMax) =>
216           response.status(Status.UnprocessableEntity).body(Webservice.Error(
217             errorCode = "brightness-outside-supported-range",
218             description = s"brightness value $value was outside the supported range: ($supportedRangeMin ... $supportedRangeMax)",
219             diagnosticInfo = None
220           ))
221       }
222     )
223   }
```

So it is almost the same as it was in TELL situation, the only difference being the **responseConverter** added - and this must be a partial function analyzing the message received from actor. If the pattern

matching fails – it is automatically understood that a reply message was “unexpected” and a proper error response is build on the webservice level. All this gets quite complicated in details because of errors handling (without errors handling it would be possible to pack all this into one for-comprehension).

Look how I implemented **akkaProcessing_Ask**:

```
349   def akkaProcessing_Ask(actorPath: String, sendMsg: Any, responseConverter: PartialFunction[Any,Response]): com.twitter.util.Future[Response] = {
350     implicit val timeout: Timeout = Config.finatraAkkaBridgeAskTimeout
351     implicit val executionContext = actorSystem.dispatcher
352
353     val futureActorRef: Future[ActorRef] = actorSystem.actorSelection(actorPath).resolveOne(Config.finatraAkkaBridgePathResolvingTimeout)
354     val handlerOfUnexpetedMsgGotFromActor: PartialFunction[Any, Response] = {case msg =>
355       response.internalServerError(Webservice.Error(
356         errorCode = "unexpected-msg-from-actor",
357         description = "server-side problem, please contact the service team",
358         diagnosticInfo = Some(s"actor path = $actorPath, received msg = $msg")
359       ))}
360
361     val responseConverterWithUnexpectedMsgHandling: Any => Response = responseConverter orElse handlerOfUnexpetedMsgGotFromActor
362
363     val scalaFutureWithAllCasesHandled: Future[Response] = (futureActorRef flatMap {actorRef => (actorRef ? sendMsg) map responseConverterWithUnexpectedMsgHandling}) recover {
364       case ex: ActorNotFound => response.notFound(Webservice.Error(
365         errorCode = "not-such-resource",
366         description = "the URI pointed to non-existing resource, this is client-side error",
367         diagnosticInfo = Some(s"request internally converted to akka actor path: $actorPath, but such actor was not found in the system")
368       ))
369       case ex: AskTimeoutException => response.internalServerError(Webservice.Error(
370         errorCode = "finatra-akka-bridge-level-akka-ask-timeout",
371         description = "server-side problem, please contact the service team; this may be a server-too-busy side efect",
372         diagnosticInfo = Some(s"actor path = $actorPath, sent msg = $sendMsg,  timeout is currently configured to: ${Config.finatraAkkaBridgeAskTimeout}")
373       ))
374     }
375
376     return TwitterFutureConverters.scalaToTwitterFuture(scalaFutureWithAllCasesHandled)(actorSystem.dispatcher)
377   }
```

Logging implementation and configuration

Slf4j and Logback

UNDER CONSTRUCTION

Akka logging

UNDER CONSTRUCTION

Finatra logging

UNDER CONSTRUCTION

Controllers logging

UNDER CONSTRUCTION

Building and running the solution

Source code repository

UNDER CONSTRUCTION

Building

UNDER CONSTRUCTION

Running

UNDER CONSTRUCTION

Testing with Postman

UNDER CONSTRUCTION

Final remarks

Scaling to "production ready"

UNDER CONSTRUCTION
(modularization of controllers, twitter server interface, mastering finacle filters)

Finacle vs Twitter Server vs Finatra

UNDER CONSTRUCTION

Finatra vs Akka HTTP vs Play

UNDER CONSTRUCTION

Actor model processing complexity

UNDER CONSTRUCTION