ACTORBASE

SPREADING ACTORS ALL OVER THE WORLD

Developer Manual

Document Information

Version	2.0.0
Editing	Francesco Agostini
	Alberto De Agostini
	Giacomo Vanin
Verification	Marco Boseggia
Approvation	Davide Trevisan
Use	External
Distribution List	ScalateKids
	Prof. Tullio Vardanega
	Prof. Riccardo Cardin

History log

Version	Author	Role	Date	Description
2.0.0	Davide Trevisan	Project Manager	2016-06-24	Document approved
1.2.0	Marco Boseggia	Verifier	2016-06-23	Verified section 3
1.1.1	Alberto De Agostini	Programmer	2016-06-22	Improvements to almost every chapter in section 3 with some updates to reflect the driver usage
1.1.0	Marco Boseggia	Verifier	2016-06-20	Verified section 2 and 3.1
1.0.3	Giacomo Vanin	Programmer	2016-06-17	Improvements to section 3.1
1.0.2	Giacomo Vanin	Programmer	2016-06-15	Added section 2
1.0.1	Francesco Agostini	Programmer	2016-05-30	Added glossary
1.0.0	Michael Munaro	Project Manager	2016-05-15	Document approved
0.11.0	Alberto De Agostini	Verifier	2016-05-15	Verified subsection 2.1.1 - Server-side general structure
0.10.1	Marco Boseggia	Programmer	2016-05-15	Improvements to section 2.1 - Installation, added subsection 2.1.1 - Server-side general structure
0.10.0	Davide Trevisan	Verifier	2016-05-15	Verified section 2.3.0.5 - Contribute and relative subsections
0.9.0	Davide Trevisan	Verifier	2016-05-15	Verified improvements in section 2.1.1 - Building from sources, build configuration for CLI and Server
0.8.3	Marco Boseggia	Programmer	2016-05-15	Added section 2.3.0.5 - Contribute and relative subsections
0.8.2	Giacomo Vanin	Programmer	2016-05-15	Improved section 2.1.1 - Building from sources, added build configuration for CLI
0.8.1	Giacomo Vanin	Programmer	2016-05-14	Improved section 2.1.1 - Building from sources, added build configuration for server
0.8.0	Alberto De Agostini	Verifier	2016-05-14	Verified sample configuration in section 2.1 - Installation
0.7.1	Andrea Giacomo Baldan	Programmer	2016-05-14	Improved section 2.1 - Installation, added sample configuration
0.7.0	Davide Trevisan	Verifier	2016-05-13	Verified section 2.1.1 - Building from sources

0.6.1	Marco Boseggia	Programmer	2016-05-13	Added section 2.1.1 - Building from sources
0.6.0	Davide Trevisan	Verifier	2016-05-13	Verified section 2.1 - Installation
0.5.1	Marco Boseggia	Programmer	2016-05-13	Added section 2.1 - Installation
0.5.0	Davide Trevisan	Verifier	2016-05-13	Verified section 2.1.1 - Building from sources
0.4.1	Marco Boseggia	Programmer	2016-05-13	Added section 2.1.1 - Building from sources
0.4.0	Davide Trevisan	Verifier	2016-05-13	Verified section 2.4.0.4 - Failure management
0.3.0	Davide Trevisan	Verifier	2016-05-13	Verified sections 2.4.0.2 - Profile management, 2.4.0.3 - Collections operations and relative subsections
0.2.2	Francesco Agostini	Programmer	2016-05-13	Added section 2.4.0.4 - Failure management
0.2.1	Andrea Giacomo Baldan	Programmer	2016-05-13	Added sections 2.4.0.2 - Profile management, 2.4.0.3 - Collections operations and relative subsections
0.2.0	Alberto De Agostini	Verifier	2016-05-12	Verified section 2.3.1 - Quick start, 2.4 - More in depth: common operations and 2.4.1 - Authentication
0.1.2	Andrea Giacomo Baldan	Programmer	2016-05-12	Added section 2.4 - More in depth: common operations and 2.4.1 - Authentication
0.1.1	Andrea Giacomo Baldan	Programmer	2016-05-12	Added section 2.3.1 - Quick start
0.1.0	Alberto De Agostini	Verifier	2016-05-10	Verified section 2: Actorbase Usage and subsection 2.3
0.0.2	Andrea Giacomo Baldan	Programmer	2016-05-10	Added section 2: Actorbase Usage and subsection 2.3
0.0.1	Andrea Giacomo Baldan	Programmer	2016-05-10	Created document structure

Contents

1 Summary **1**
1.1 Document purpose . . . 1
1.2 Product purpose . . . 1
1.3 Glossary . . . 1

2 Default users **2**

3 Booting Actorbase **3**

4 Actorbase usage **4**
4.1 Installation . . . 4
4.1.1 Minimum requirements . . . 4
4.1.2 Installation steps . . . 4
4.1.3 Security . . . 7
4.1.4 Server-side general structure . . . 8
4.1.5 Building from sources . . . 8
4.2 ActorbaseDriver: Using Actorbase from inside a Scala source . . . 10
4.2.1 Quick Start: a generic example . . . 10
4.3 More in depth: common operations . . . 12
4.3.0.1 Authentication . . . 12
4.3.0.2 Profile management . . . 12
4.3.0.2.1 Credential rules . . . 12
4.3.0.3 Collections operations . . . 12
4.3.0.3.1 Contributor management . . . 16
4.3.0.3.2 Administrative operations . . . 17
4.3.0.4 Failure management . . . 17
4.3.0.5 Contribute . . . 18
4.3.0.5.1 Issues . . . 18
4.3.0.5.2 Troubleshooting . . . 18

A Glossary **19**
A.1 A . . . 19
Actor . . . 19
A.2 B . . . 19
Boilerplate . . . 19
A.3 C . . . 19
Clientactor . . . 19
Cluster . . . 19
Collection . . . 19
A.4 M . . . 19
Main . . . 19
Marshalling . . . 19
A.5 N . . . 19

NoSql 19
A.6 O 20
Override 20
A.7 S 20
Scale-out 20
Shell 20
Snippet 20
A.8 V 20
Vararg 20

1 Summary

1.1 Document purpose

This document aims to describe the procedures and methods to use in order to take advantage of **Actorbase** features, including installation steps and configuration available to users.

1.2 Product purpose

The purpose of the project is the realization of a key-value type NoSQL$_G$ database, massively concurrent and oriented to the management of heavy load of data by using the actor$_G$ model.

1.3 Glossary

Every technical word, some words that can be misunderstood and all of acronyms are defined in the glossary in the appendix A. Every word that is in the glossary will be marked with a subscript ”G”.

2 Default users

When *actorbase* is launched for the first time it creates two default users:

- **admin**: this user has admin privileges, he can add and remove users, reset the password of an user to "Actorb4se" and he is the only one who has access to all of the collections of the database;
- **anonymous**: this user is the default non-admin user. Everything that has been done by an anonymous user will not be persisted.

Both of these users will be created with the default password "Actorb4se". It is recommended for the admin to change his password after the first login.

3 Booting Actorbase

To successfully boot *Actorbase* you need to choose one of the two following procedures:

- start up the first seed node registered in the configuration file first;
- set up the minimum number of nodes required to the actual number of nodes that you will be using.

4 Actorbase usage

4.1 Installation

4.1.1 Minimum requirements

In addition to JVM version 8, for a pleasing experience **Actorbase** requires at minimum:

- RAM:1 GB;
- Disk space: 512MB;
- CPU: x64 - 1.2GHz;

4.1.2 Installation steps

The system is basically constituted of two JAR (Java Archive) shipped with all dependencies libraries, there are just a few steps to follow to have the system running and ready to receive commands:

- run `actorbase.jar` in a shell$_G$;
- run `actorbasecli.jar` in a second shell$_G$.

That's really all for a basic usage, and the procedure is the same for *Windows* 10, *Ubuntu 14.04 LTS*, *OSX - El Capitan* and following versions:

```
$ sbt assembly
$ ./target/scala-2.11/actorbase.jar --config=path/to/config.cfg
```

Figure 1: Actorbase: build binaries with custom configuration

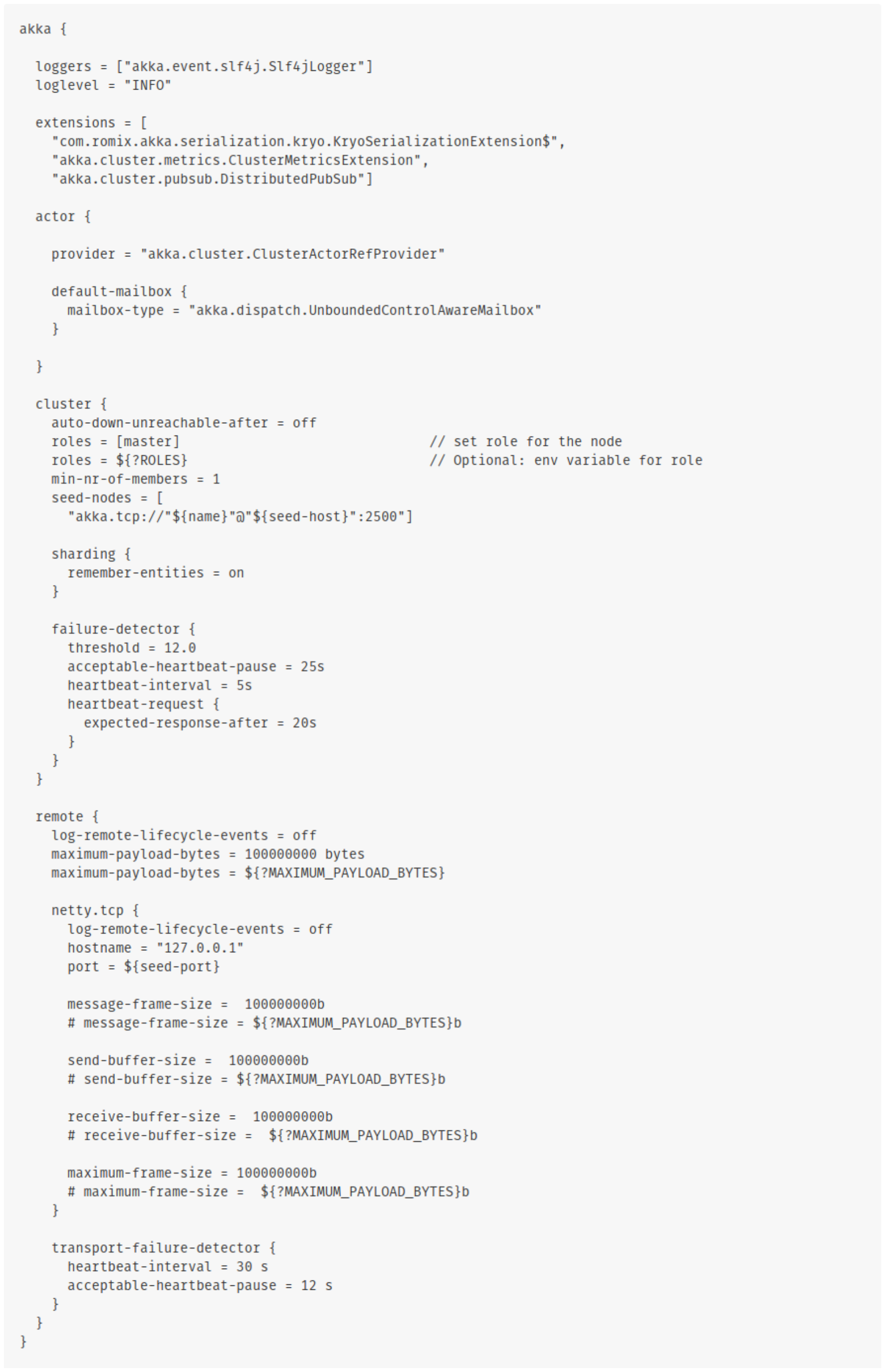
The *–config=path/to/config.cfg* is optional. Once the fat-jar is generated, it is really easy to run an instance of Actorbase on every node of the cluster, giving a custom (optionally) configuration file:

```
$ java -Dconfig.file=/dir/dir/reference.conf -jar actorbase.jar
```

Figure 2: Actorbase: build binaries with custom configuration

Configuration sample

Being an akka based application, it is possible to customize and tweak all aspects of the actor model engine by providing a custom *application.conf*, as explained at reference.conf. In addition to the standard configuration provided for akka applications, there are some attributes specific of Actorbase:



```
akka {

  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "INFO"

  extensions = [
    "com.romix.akka.serialization.kryo.KryoSerializationExtension$",
    "akka.cluster.metrics.ClusterMetricsExtension",
    "akka.cluster.pubsub.DistributedPubSub"]

  actor {

    provider = "akka.cluster.ClusterActorRefProvider"

    default-mailbox {
      mailbox-type = "akka.dispatch.UnboundedControlAwareMailbox"
    }

  }

  cluster {
    auto-down-unreachable-after = off
    roles = [master]                               // set role for the node
    roles = ${?ROLES}                              // Optional: env variable for role
    min-nr-of-members = 1
    seed-nodes = [
      "akka.tcp://"${name}"@"${seed-host}":2500"]

    sharding {
      remember-entities = on
    }

    failure-detector {
      threshold = 12.0
      acceptable-heartbeat-pause = 25s
      heartbeat-interval = 5s
      heartbeat-request {
        expected-response-after = 20s
      }
    }
  }

  remote {
    log-remote-lifecycle-events = off
    maximum-payload-bytes = 100000000 bytes
    maximum-payload-bytes = ${?MAXIMUM_PAYLOAD_BYTES}

    netty.tcp {
      log-remote-lifecycle-events = off
      hostname = "127.0.0.1"
      port = ${seed-port}

      message-frame-size =  100000000b
      # message-frame-size = ${?MAXIMUM_PAYLOAD_BYTES}b

      send-buffer-size =  100000000b
      # send-buffer-size = ${?MAXIMUM_PAYLOAD_BYTES}b

      receive-buffer-size =  100000000b
      # receive-buffer-size =  ${?MAXIMUM_PAYLOAD_BYTES}b

      maximum-frame-size = 100000000b
      # maximum-frame-size =  ${?MAXIMUM_PAYLOAD_BYTES}b
    }

    transport-failure-detector {
      heartbeat-interval = 30 s
      acceptable-heartbeat-pause = 12 s
    }
  }
}
```

Figure 3: Actorbase: default configuration sample

Default configuration sample

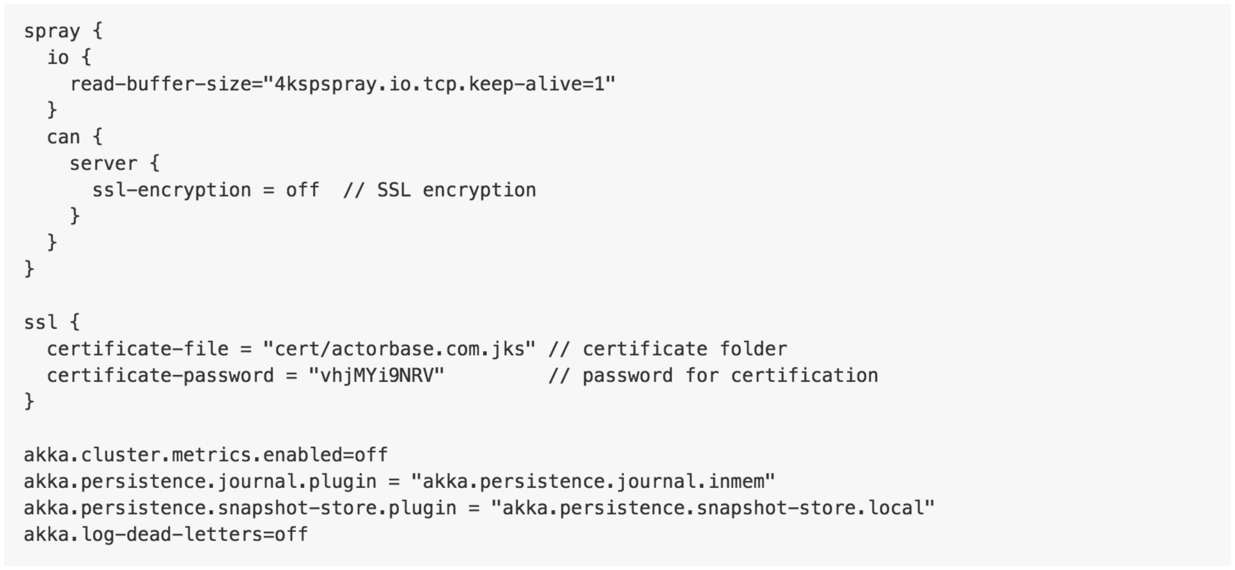
```
// storekeeper actors configuration
storekeepers {
  role = ""                // Optional: set role for storekeepers
  max-instances = 100000   // max instance inside the cluster
  instances-per-node = 20  // max instance per node, define storekeepers per collection
  size = 256               // number of keys stored per storekeeper
  save-method = "snapshot" // save method, can be 'snapshot' or 'onchange'
  snapshot-conf {             // snapshot configuration, mandatory for 'snapshot' save-method
    first-snapshot-after = 20 // ignored with 'onchange' policy
    snapshot-every = 50       // sets how many seconds between every save
  }
}

// storage configuration
persistence {
  save-folder = "actorbasedata/"      // folder where all data will be saved
  encryption-algorithm = "AES"        // encryption algorithm
  encryption-key = "hsujHu6UshHJslkV" // AES encryption key
}

name = actorbase  // actorsystem name
name = ${?NAME}   // Optional: env variable for actorsystem name

listen-on = "127.0.0.1"   // address listening for connections
listen-on = ${?LISTEN_ON} // Optional: env variable for listen address

exposed-port = 9999             // port open to connections
exposed-port = ${?EXPOSED_PORT} // Optional: env variable for port

seed-host = "127.0.0.1"   // base seed-node
seed-host = ${?SEED_HOST} // Optional: env variable for base seed-node

seed-port = 2500          // base seed-port
seed-port = ${?SEED_PORT} // Optional: env variable for base seed-port

shard-number = 40               //shard number for cluster sharding, rule of thumb: number of nodes x 10
shard-number = ${?SHARD_NUMBER} //Optional: env variable for shard number

// akka.cluster.use-dispatcher = cluster-dispatcher // cluster dispatcher, can be used to tune akka based
                                                    // on the system running the application
// cluster-dispatcher {
//  type = "Dispatcher"
//  executor = "fork-join-executor"
//  fork-join-executor {
//    parallelism-min = 2
//    parallelism-factor = 1.0
//    parallelism-max = 4
//  }
//   throughput = 100
//}
```

Figure 4: Actorbase: Actorbase configuration sample

For *roles, message-frame-size, send-buffer-size, receive-buffer-size* and *maximum-payloads-bytes* is possible to use environment variables just by setting them in the node where the server has been started.

4.1.3 Security

Actorbase supports also crypto communications.



```
spray {
  io {
    read-buffer-size="4kspspray.io.tcp.keep-alive=1"
  }
  can {
    server {
      ssl-encryption = off  // SSL encryption
    }
  }
}

ssl {
  certificate-file = "cert/actorbase.com.jks" // certificate folder
  certificate-password = "vhjMYi9NRV"         // password for certification
}

akka.cluster.metrics.enabled=off
akka.persistence.journal.plugin = "akka.persistence.journal.inmem"
akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
akka.log-dead-letters=off
```

Figure 5: Actorbase: SSL secure communication

4.1.4 Server-side general structure

As previously stated, **Actorbase** is an application conceived to be used in a distributed fashion, balancing load between multiple nodes as shown below:

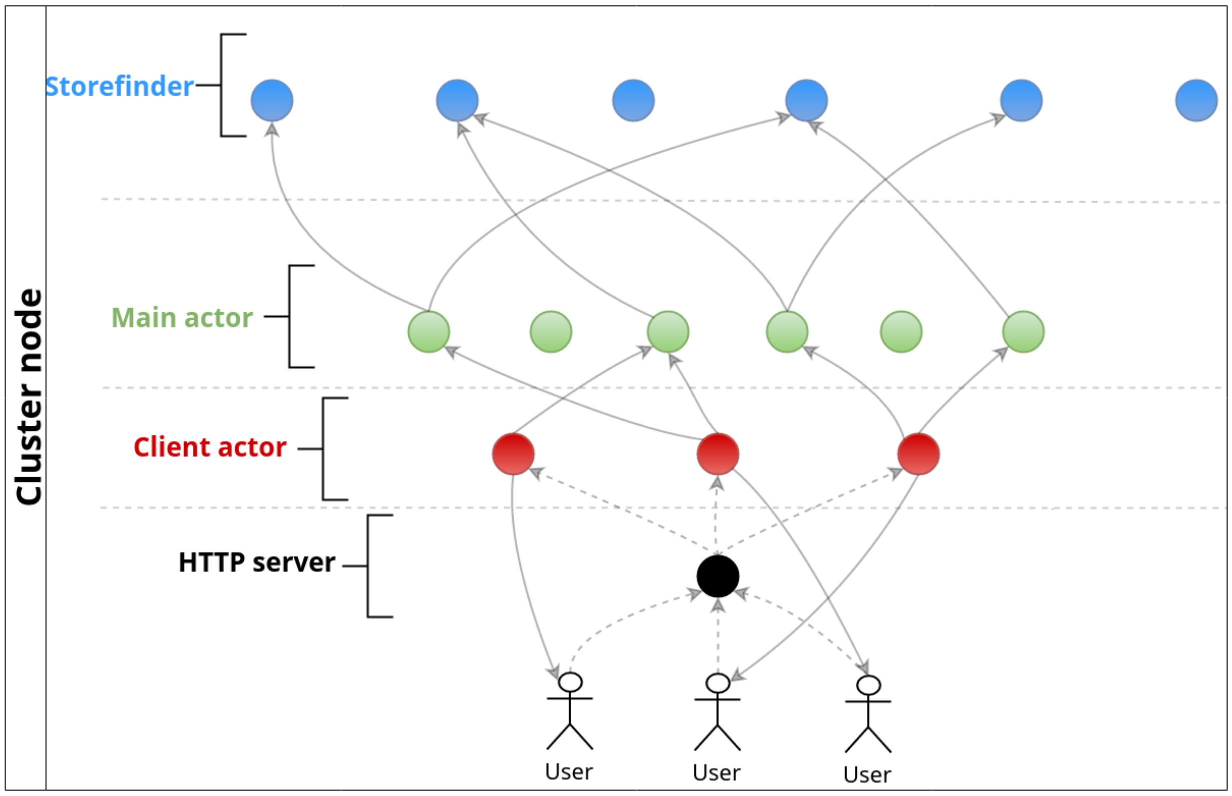


Figure 6: Actorbase: load-balancing general view

For every incoming connection a dedicated actor$_G$, named clientactor$_G$, is spawned; it is responsible for all incoming requests from the client that is connected, and his main role is to forward these requests directly to the main$_G$ actors$_G$ equally distributed across the nodes of a cluster$_G$ using cluster-sharding extension of Akka library. In this way, with main$_G$ actors$_G$ breeding their own hierarchy of subordinates, all incoming data is spread across the cluster$_G$ network.

4.1.5 Building from sources

Actorbase was conceived and developed using SBT (Simple Building Tool) as building and dependencies manager tool making the compilation and building from sources extremely easy to achieve just by adding a `build.sbt` file inside a Maven-like folder tree:

Server side

```
name := "Actorbase-Server"
version := "1.0"
scalaVersion := "2.11.8"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.4.6",
  "com.typesafe.akka" %% "akka-testkit" % "2.4.6",
  "com.typesafe.akka" %% "akka-cluster-sharding" % "2.4.6",
  "com.typesafe.akka" %% "akka-cluster-tools" % "2.4.6",
  "com.typesafe.akka" % "akka-cluster-metrics_2.11" % "2.4.6",
  "com.typesafe.akka" %% "akka-slf4j" % "2.4.6",
  "com.typesafe" % "config" % "1.2.1",
  "org.scalatest" % "scalatest_2.11" % "2.2.6" % "test",
  "io.spray" %% "spray-can" % "1.3.3",
  "io.spray" %% "spray-routing" % "1.3.3",
  "io.spray" %% "spray-json" % "1.3.2",
  "org.mindrot" % "jbcrypt" % "0.3m",
  "com.github.t3hnar" % "scala-bcrypt_2.10" % "2.6",
  "com.github.romix.akka" %% "akka-kryo-serialization" % "0.4.1",
  "org.apache.maven.plugins" % "maven-shade-plugin" % "2.4.3",
  "ch.qos.logback" % "logback-classic" % "1.1.3"
)

javaOptions ++= Seq("-Xmx2048m")
```

Figure 7: Actorbase: build.sbt sample to build from sources

CLI

```
name := "Actorbase-CLI"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies ++= Seq(
  "com.typesafe" % "config" % "1.2.1",
  "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.2",
  "org.scala-lang" % "jline" % "2.11.0-M3",
  "org.scalatest" % "scalatest_2.11" % "2.2.6" % "test",
  "org.scalaj" %% "scalaj-http" % "2.3.0",
  "org.scala-lang.modules" %% "scala-pickling" % "0.10.1",
  "org.json4s" %% "json4s-native" % "3.3.0",
  "org.json4s" %% "json4s-jackson" % "3.3.0",
  "io.spray" %%  "spray-json" % "1.3.2")
```

Figure 8: Actorbase-CLI: build.sbt sample to build from sources

As shown in the image, there are some external libraries needed to build the system:

- **Spray:** (http://spray.io/) An open source toolkit for building REST/HTTP-based integration layers on top of Scala and Akka; mainly used for the server component in order to expose communication API and to Marshall$_G$ database contents.
- **scalaj-http:** (https://github.com/scalaj/scalaj-http) A fairly simple http library for Scala, used in the driver component in order to establish a communication with the REST API server;
- **json4s:** (http://json4s.org/) Used in the driver component to expose a pretty formatted JSON (JavaScript Object Notation) version of the data retrieved from the server;
- **scala-bcrypt:** (https://github.com/t3hnar/scala-bcrypt) Scala version of the encryption library Bcrypt, used on the server side to store hashes of sensitive data;
- **Akka:** (http://akka.io/) A toolkit and runtime environment to build highly concurrent, distributed, and resilient message-driven applications on the JVM; it is the core of the system and it is used in the server-side component.

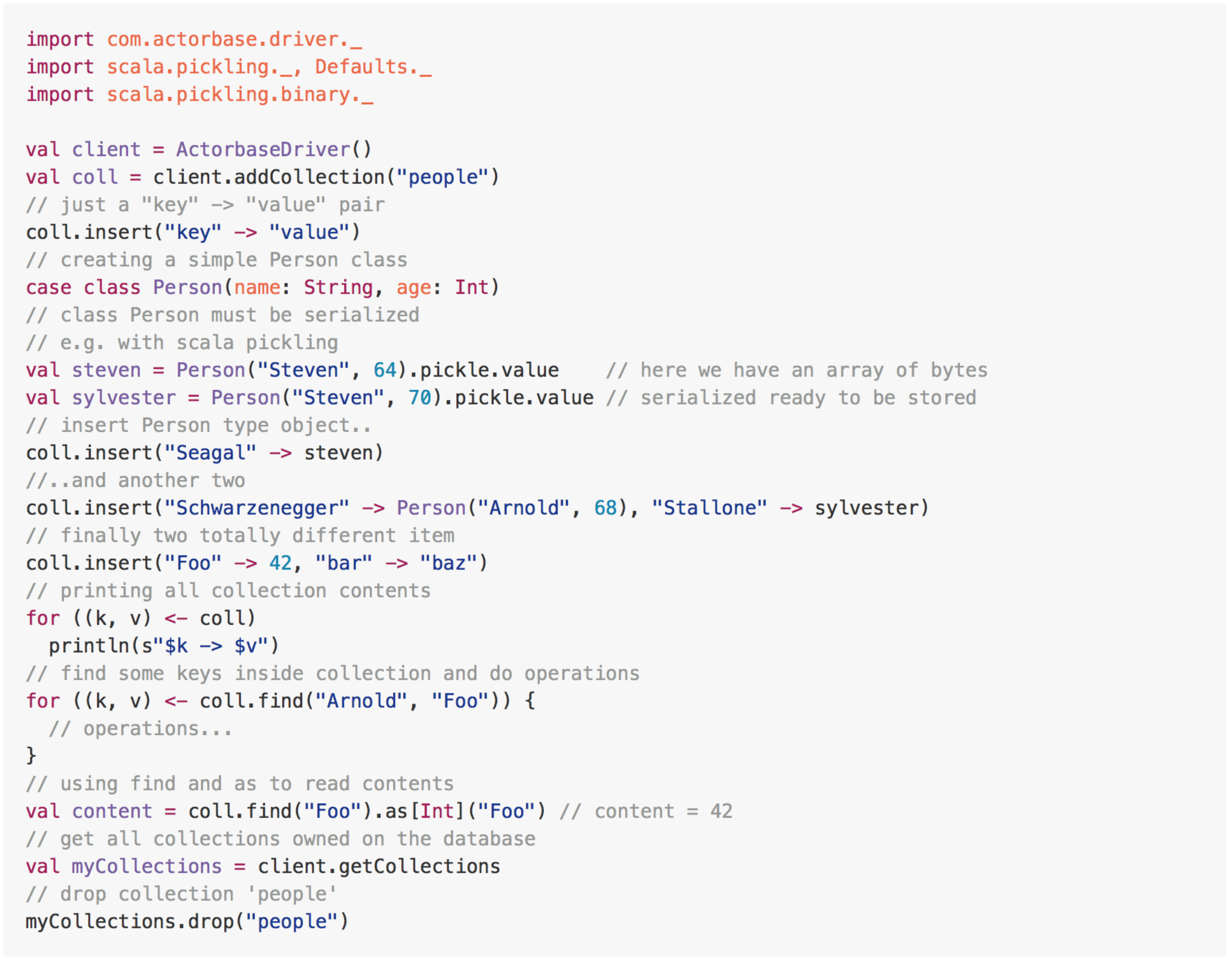
4.2 ActorbaseDriver: Using Actorbase from inside a Scala source

Actorbase is essentially an actor$_G$ system built upon a server component that exposes some API forming a RESTful web service. To facilitate the use from inside a Scala source, a driver has been developed in order to allow the usage of all features that **Actorbase** provides.

4.2.1 Quick Start: a generic example

The most common operations include:

- Authentication;
- Profile management operations;
- Collections operations;
- Administrative operations.



```scala
import com.actorbase.driver._
import scala.pickling._, Defaults._
import scala.pickling.binary._

val client = ActorbaseDriver()
val coll = client.addCollection("people")
// just a "key" -> "value" pair
coll.insert("key" -> "value")
// creating a simple Person class
case class Person(name: String, age: Int)
// class Person must be serialized
// e.g. with scala pickling
val steven = Person("Steven", 64).pickle.value    // here we have an array of bytes
val sylvester = Person("Steven", 70).pickle.value // serialized ready to be stored
// insert Person type object..
coll.insert("Seagal" -> steven)
//..and another two
coll.insert("Schwarzenegger" -> Person("Arnold", 68), "Stallone" -> sylvester)
// finally two totally different item
coll.insert("Foo" -> 42, "bar" -> "baz")
// printing all collection contents
for ((k, v) <- coll)
  println(s"$k -> $v")
// find some keys inside collection and do operations
for ((k, v) <- coll.find("Arnold", "Foo")) {
  // operations...
}
// using find and as to read contents
val content = coll.find("Foo").as[Int]("Foo") // content = 42
// get all collections owned on the database
val myCollections = client.getCollections
// drop collection 'people'
myCollections.drop("people")
```

Figure 9: Driver usage: Generic example

These are some of the most common operations that the driver allows the user to do, there is not too much boilerplate code, and it is possible to make operations directly on objects representing **Actorbase** collections$_G$ in a purely OOP (Object Oriented Programming) fashion.
As explained in the snippet image these objects also retain some of the common functional constructs (e.g. foreach), so it's possible to apply custom functions to every item of the collection.

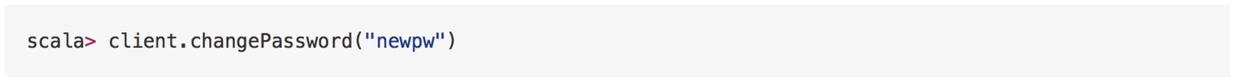
4.3 More in depth: common operations

4.3.0.1 Authentication

Although in the future there may be the possibility to make some operations as an anonymous user, at the current state the only possible action that a non-logged-in user can do with `ActorbaseDriver` is the authentication. This method sends a login request to the server-side of the system and, according to the response, it will return an `ActorbaseService` or an `ActorbaseAdminServices`. These two objects give the methods to actually make all operations on the database, the main difference is that `ActorbaseAdminServices` allows to do some high-level administrative operations on the users and the collections$_G$ of the system.
Authentication method could raise **WrongCredentialExc** exception, in case wrong credentials has been sent to the server.

4.3.0.2 Profile management

ActorbaseDriver provides a method to modify the current password, associated to an user profile. e.g.:



```
scala> client.changePassword("newpw")
```

Figure 10: Driver usage: insertion methods

This method could raise:

- **WrongPasswordExc** In case the user made a mistake inserting the current password;
- **WrongNewPasswordExc** In case the new password does not meet the **Actorbase** password rules.

4.3.0.2.1 Credential rules

Usernames can not be an empty string or have any blank spaces, every password must contain at least one uppercase and one lowercase letter, one digit and must be at least 8 characters long.

4.3.0.3 Collections operations

As previously explained, operations on the collections$_G$ can be done in a purely OOP fashion, this means that all methods that return an `ActorbaseCollection`, give the possibility to call all collection-management methods:

- `listCollection`, lists all collections$_G$ contained in the database, these are the collections$_G$ owned by the profile of the applicant, including the ones he is a contributor of.
- `addCollection` method:
  Create a new collection$_G$ into **Actorbase**, accepting a `String` representing the name of the new collection$_G$; e.g.:

```
scala> client.addCollection("consumers")
res0: com.actorbase.driver.data.ActorbaseCollection

// as contributor to "contributee.consumer" (or admin)
scala> client.addCollection("consumers", "contributee")
res1: com.actorbase.driver.data.ActorbaseCollection
```

Figure 11: Driver usage: add collection methods

Add collection$_G$ method could raise:

- **CollectionAlreadyExistsExc:** In case the name of the collection$_G$ is already taken by another collection$_G$ in **Actorbase**;
- **UndefinedCollectionExc:** In case the user tried to insert a new collection$_G$ with an empty name.

- `insert` methods:
These methods come in two variants: the first one accepting a vararg$_G$ of key-value tuple and the second one accepting an `ActorbaseObject` instance; e.g.:

```
scala> client.getCollection("people").insert("keyOne" -> "valueOne", "keyTwo" -> 42)
res0: com.actorbase.driver.data.ActorbaseCollection

scala> client.getCollection("people").insert(ActorbaseObject("obj" -> "inserting with object"))
res1: com.actorbase.driver.data.ActorbaseCollection
```

Figure 12: Driver usage: insertion method

```
scala> client.insert("people", false, "key" -> "value") // insert without update to collection "people"

scala> client.insertTo("people", false, "key" -> "value")("mike")
// insert without update to collection "people" owned by user mike
// just in case of being a contributor with write permissions, or admin
```

Figure 13: Driver usage: alternative insertion method

Insert methods could raise:

- **DuplicateKeyExc:** In case of a key is already taken inside the collection$_G$.

- `remove` methods:
There are two variants of this method too, the usage is analogous to insert, e.g.:

```
scala> client.getCollection("people").remove("people", "keyOne")
res0: com.actorbase.driver.data.ActorbaseCollection
```

Figure 14: Driver usage: remove methods

- `find` method:
  This method accepts one or more `Strings` representing keys to be retrieved from the collection$_G$; once again the usage is similar to the insert and remove methods, e.g.:

```
scala> client.find("keyOne", "people")
res0: com.actorbase.driver.data.ActorbaseObject
```

Figure 15: Driver usage: find method

- `drop` methods:
  Again this method has multiple variants, there is the `dropCollections` that can be used to wipe out all the database, e.g.:

```
scala> client.dropCollections

scala> client.getCollections.drop

scala> client.dropCollections("customers", "people")
```

Figure 16: Driver usage: dropCollections method

the `drop` inside an `ActorbaseCollection` object:

```
scala> client.dropCollections

scala> client.getCollections.drop

// as admin
scala> client.dropCollections("customers", "people")
```

Figure 17: Driver usage: drop method

and finally `drop` inside an `ActorbaseCollectionMap`, it takes a vararg of `String` representing a sequence of collections$_G$ to be removed e.g.:

```
scala> auth.getCollections.drop
res0: Boolean = true
```

Figure 18: Driver usage: drop method

- printing collections$_G$:
  All objects returned by `ActorbaseDriver` have an override$_G$ to the method `toString` that returns a pretty formatted JSON (JavaScript Object Notation) `String`, e.g.:

```
scala> println(client.getCollection("people"))
{
  "owner": "foo",
  "contributors": {},
  "collectionName": "people",
  "data": {
    "bar": "baz",
    "Foo": 42,
    "Seagal": {
        "type": "Person",
        "name": "Steven",
        "age": 64
    },
    "Schwarzenegger": {
        "type": "Person",
        "name": "Arnold",
        "age": 68
     },
    "Stallone": {
        "type": "Person",
        "name": "Sylvester",
        "age": 70
     }
  }
}
```

Figure 19: Driver usage: printing a single collection

it's also possible to print all collections owned:

```
scala> client.getCollections.foreach(println)
```

Figure 20: Driver usage: printing multiple collections

- `count` items inside a collection$_G$ e.g.

```
scala> println(client.getCollection("people").count)
res0: Int = 3
```

Figure 21: Driver usage: drop method

- `importData`, method that allows to import a (possibly) large number of collections$_G$ directly into **Actorbase**

```
scala> client.importData("foo/bar/baz.json")
```

Figure 22: Driver usage: import from file method

  This method could raise:
  - **UndefinedFileExc:** in case the file has not been found at the given path in the filesystem;
  - **MalformedFileExc:** in case the file is not in JSON format or it is not a valid JSON.
- `exportData`, method that allows to export one or more collections$_G$ to a specified path on the filesystem

```
scala> client.exportData("foo/bar/customers_people.json", "customers", "people")

scala> client.exportData("foo/bar/all_collections.json")
```

Figure 23: Driver usage: export to file method

4.3.0.3.1 Contributor management

Each user in **Actorbase** has the possibility to add and remove contributors to his collections$_G$, a user can be added with two levels of permissions:

- read, means that a user can only make read operations on the collection$_G$ without modifying it;
- read-write, means that a user can make read and write operations on the collection$_G$.

e.g.:

```
scala> val fooCollection = client.getCollection("foo")

scala> fooCollection.addContributor("aFriend", true) // add

scala> fooCollection.removeContributor("aFriend", true) // remove

scala> client.addContributor("aFriend", "people", false)
// aFriend is now contributor of collection "people" with read-only permissions

scala> client.addContributor("aFriend", "people", false, "mike")
// administrator operation: aFriend is now contributor of collection
// "people" with read-only permissions to the collection owned by mike
```

Figure 24: Driver usage: contributor management

4.3.0.3.2 Administrative operations

Actorbase expects two kind of users, common and administrator. The last one has some privileges over the others, such as a wider range of collections$_G$ (all database) with read-write permissions, and capabilities of adding and removing users to and from the system, and finally resetting the password of given users.

```
scala> client.addUser("aUser")

scala> client.removeUser("aUser")

scala> client.resetPassword("anotherUser")
```

Figure 25: Driver usage: administrative operations

Administrative operations could raise some exceptions:

- **UndefinedUsernameExc:** In case of addition of a new user with an empty username;
- **UsernameNotFoundExc:** In case the username inserted in the remove command is not found.

4.3.0.4 Failure management

In addition to all exceptions that could be raised, there is one error that is shared by every method and that is in case of a communication error, which can be caused by not receiving a response from the server in a predetermined time or if the destination address is unreachable, the driver will launch a timeout connection exception.

4.3.0.5 Contribute

4.3.0.5.1 Issues

Actorbase is hosted on github at the url `https://github.com/ScalateKids` divided in two repositories:

- **Server-side**
  - **Issue Tracker:** `https://github.com/ScalateKids/Actorbase-Server/issues`
  - **Source code:** `https://github.com/ScalateKids/Actorbase-Server`
- **CLI**
  - **Issue Tracker:** `https://github.com/ScalateKids/Actorbase-Client/isssues`
  - **Source code:** `https://github.com/ScalateKids/Actorbase-Client`

4.3.0.5.2 Troubleshooting

If you are having issues or strange malfunctions, please let us know at scalatekids@gmail.com.

A Glossary

A.1 A

Actor: A computational entity that, in response to a message it receives, can send messages to other actors, create new ones, change its behaviour for the next message incoming.

A.2 B

Boilerplate: A section of code that has to be included in many places with little to no alteration.

A.3 C

Clientactor: An actor dedicated exclusively to handle incoming request from clients connected from outside the system.

Cluster: A set of loosely or tightly connected computers that work together, so that they can beviewed as a single system.

Collection: A data-structure constituted by a list of key-value items.

A.4 M

Main: An actor dedicated to the routing of incoming data to subordinated actors and to the management of the latter.

Marshalling: The act of transforming the memory of an object to a data format suitable for storage or transmission, e.g. serialization.

A.5 N

NoSql: A NoSQL database provides a mechanism for storage and retrieval of data which is modeled in means other than the tabular relations used in relational databases. Key-value (KV) stores use the associative array (also known as a map or dictionary) as their fundamental data model.

A.6 O

Override: Specific implementation of a method in a subclass that is already provided by one of its superclasses.

A.7 S

Scale-out: Method of adding more resources by adding more nodes to the cluster.

Shell: A command-line interpreter.

Snippet: A programming term for a small region of reusable code.

A.8 V

Vararg: A sequence of arguments.