AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF COMPUTER SCIENCE,
ELECTRONICS AND TELECOMMUNICATIONS

TOOL FOR MONITORING OF DISTRIBUTED APPLICATIONS IN AKKA TOOLKIT

TECHNICAL DOCUMENTATION

Michał Ciołczyk
Mariusz Wojakowski

TUTOR
Maciej Malawski, Ph.D.

KRAKÓW 2016

Table of Contents

1 Project domain

1.1 The actor model in Akka

1.1.1 How does an actor look like?

Actors in Akka toolkit are classes which extend *Actor* trait and implement the *receive* method. The *receive* method should define a series of case statements, that defines which messages actor can handle, using pattern matching available in Scala language. Every case should be provided with the implementation of behaviour when the message is received. Below is an example of simple actor:

```
import akka.actor.Actor

class ExampleActor extends Actor {
  def receive = {
    case "test" => println("received test")
    case _      => println("received unknown message")
  }
}
```

The *receive* method has the type *PartialFunction[Any, Unit],* which means that you don't have to provide a pattern match for all messages. However, if you want to handle unknown messages then you need to have a default case as in the example above. The return type of the behaviour is *Unit:* if the actor shall reply to the received message then this must be done explicitly by sending a new message.

For more information about actors in Akka, please see the official Akka library documentation [1].

1.1.2 More complex example

It's a good practice to provide factory methods on the companion object of each actor which helps keeping the creation of Props very close to the actor definition (Props is a configuration object used in creating an Actor). Another advantage is that this method can perform type check on compilation, which is very useful from the developer point of view:

```
object DemoActor {
  def props(magicNumber: Int): Props = Props(new DemoActor(magicNumber))
}

class DemoActor(magicNumber: Int) extends Actor {
  def receive = {
     case x: Int => sender() ! (x + magicNumber)
  }
```

```
}
class SomeOtherActor extends Actor {
  // Props(new DemoActor(42)) would not be safe
  val actor = context.actorOf(DemoActor.props(42), "demo")
  // ...
}
```

Another good practice is to declare messages an actor can receive in companion object. In this approach, the developer can see which messages he needs to handle. In other words, the companion object defines the contract for actors or their protocol just like an interface defines the contract for the classes that implements it in object-oriented programming.

```
object ExampleActor {
  case class Greeting(from: String)
  case object Goodbye
}

class ExampleActor extends Actor with ActorLogging {
  import ExampleActor._

  def receive = {
    case Greeting(greeter) => log.info(s"I was greeted by $greeter.")
    case Goodbye           => log.info("Someone said goodbye to me.")
  }
}
```

1.1.3 Messages

In Akka messages can be any kind of object - there's only one constraint: they have to be immutable. Scala can't enforce immutability so you need to take care of this by yourself - this is to avoid the shared mutable state trap, as described in the official Akka documentation [2]. Primitives like String or Boolean are always immutable. The recommended approach is to use Scala case classes which work very well with pattern matching and greatly improve readability of code.

Here is an example:

```
//the case class definition
case class Message(from: String)

//creating a new case class message
val message = Message("Bob")
```

1.1.4 Sending messages

Sending messages in Akka toolkit is possible through one of the following methods:

- ! - 'fire-and-forget', send a message asynchronously and return immediately. Also known as *tell.*
- ? - sends a message asynchronously and returns a *Future.* Also known as *ask.*

Tell - 'fire-and-forget'

Sending messages using this method is very simple and looks exactly like in Erlang language:

```
actorRef ! message
```

It's the preferred way of passing messages. Also if invoked from within an actor, the sending actor reference will be implicitly passed along with the message and receiving actor can get it from *sender()* method - available in *ActorRef* class. It's very handful - developers don't need to explicitly pass sending actor reference in messages which increase readability of code.

Ask - 'send-and-receive future'

The *ask* pattern allows actor to ask another of information and get the return value through the `Future` object. To send a message this way, you need to provide a `Timeout` object, which specifies how long actor will be waiting before throwing `AskTimeoutException`. Below is an example of simple usage of that pattern:

```
case object AskMessage

class TestActor extends Actor {
  def receive = {
    case AskMessage =>
              sender ! "Returned value"
    case _ => println("Unexpected message")
  }
}

implicit val timeout = Timeout(5 seconds)
val future = myActor ? AskMessage
val result = Await.result(future, timeout.duration).asInstanceOf[String]
println(result)
```

For a complete example using the ask sending messages pattern, please see Alvin Alexander's blog post [3].

1.1.5 Akka Remote

Everything in Akka is designed to work in distributed manner: all interactions between actors use message passing and are asynchronous. As a result all functions are equally available when running your application on single JVM or cluster consisting of more machines. Success of this implementation is based on design: developers started from remote solution as a base environment for running applications and then go to local using optimization. Different approach - generalization from local to remote - is described in documentation as “bound to fail”. For more arguments and detailed discussion see paper “A Note on Distributed Computing” [4].

For more information about Akka Remote, please see official Akka documentation [5].

Usage in project

The Akka remote package is available as a separate jar file. Developer needs to provide following dependency:

```
"com.typesafe.akka" %% "akka-remote" % "2.3.9"
```

Also minimal configuration is necessary. It allows us to enable remote capabilities along with setting basic values, like hostname and port.

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
  }
}
```

This package in simple words allows communication between actors in different actor systems. In our solution it is necessary because our collector exists outside of the application actor system and we want to gather data using message passing semantics.

For more information regarding Akka please see the official Akka documentation [6].

1.2 Aspect-oriented programming

Aspect-oriented programming is a paradigm that aims to increase modularity of code by separating different concerns. It's done by providing additional behavior into existing code using *an advice* and specify exactly which *pointcuts* will be affected. This mechanism allows to add behaviors that "cut across" different abstractions in the application, choose what parts of system we want to modify and encapsulate them in "unit of code/entity". Very good example for cross-cutting concern is logging functionality: it isn't central to the business logic but every part of the application need to log some information, e.g. 'log all functions that set fields in object - method name starts with *"set"*. Using aspect-oriented programming you can easily specify which methods you want to instrument - a pointcut - and how - an advice. An example of instrumenting the code can be seen in Figure 1.

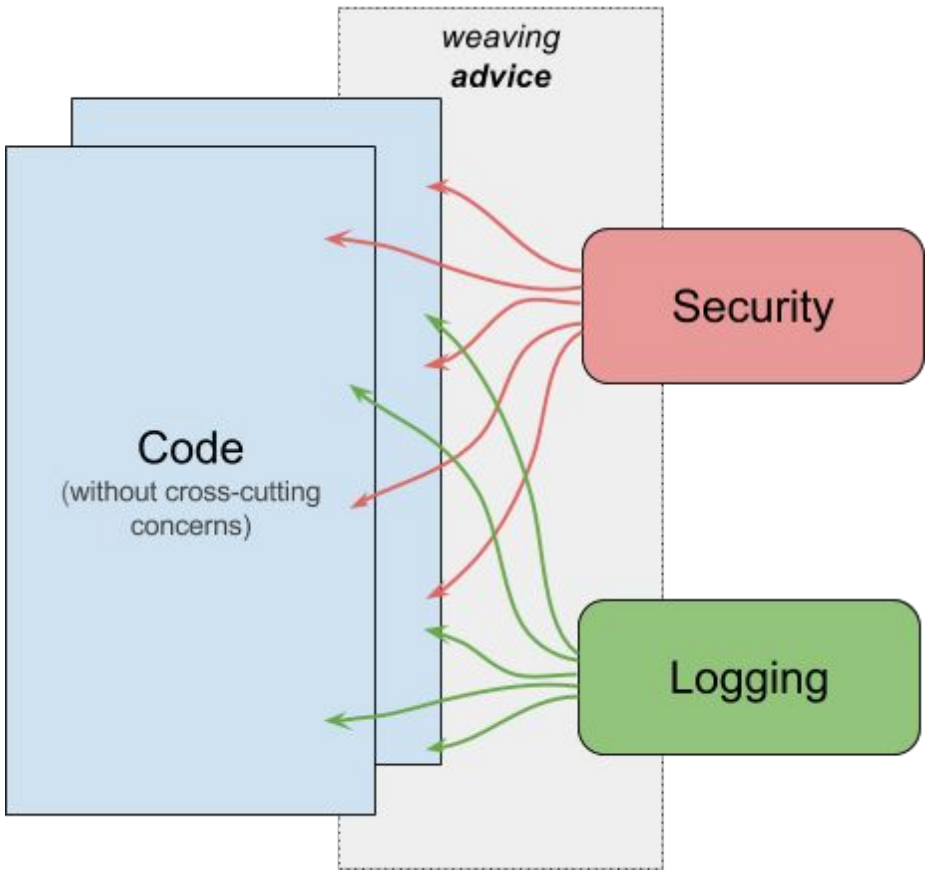

Figure 1. The main idea of aspect-oriented programming

For more information about aspect-oriented programming please see the Wikipedia article [7].

1.2.1 AspectJ

AspectJ is an aspect-oriented programming extension created for Java language. It's available both stand-alone version and integrated into Eclipse IDE. AspectJ has become widely known because of simplicity and ease of use for the developers.

For more information about AspectJ, please see the Wikipedia article [8].

AspectJ syntax

AspectJ allows to write programs using Java language syntax or specifically prepared constructions called *aspects.* Below are presented basic structures:

- **pointcuts** - helpers that allow programmers to specify which join-points want to instrument. (A join-point is well defined moments in the execution of program, e.g. function call or variable access.)

```
pointcut setPointcut(): execution (void set*(..));
```

Every method which name starts with *'set'* and return type is *void* can be weaved using this *setPointcut().*

- **advice** - encapsulate code to run and specify a joint-point. Advice can be weaved before, after or around pointcuts.

```
before (): setPointcut(): {
       Logger.debug("before 'set' method");
}
```

Annotation based syntax

Another method to create aspect is to use annotations. It's very similar to earlier mention AspectJ flavoured syntax and it's done by placing pointcut specifiers expressions in annotations, which are available in Java API from version 1.5.

It's very useful when you don't want to use AspectJ syntax because of requirements and introducing unnecessary complexity into your project.

The pointcut:

```
pointcut setPointcut(): execution (void set*(..));
```

can be written as:

```
@Pointcut("execution(void *.set*(..))")
void setPointcut() {}
```

And the advice:

```
before (): setPointcut(): {
    Logger.debug("before 'set' method");
}
```

can be written as:

```
@Before("setPointcut()")
public void beforeSetPointcut() {
    Logger.debug("before 'set' method");
}
```

Implementation

Instrumentation using aspect-oriented programming can be implemented in multiple ways, e.g. source-weaving, bytecode-weaving or runtime-weaving. It depends on the user requirements and accessibility to source code. Source-weaving is good way to provide best performance but it's necessary to have access to code you want to instrument. Bytecode-weaving allows to instrument compiled class. Runtime-weaving is very similar to the previous one - it enables to instrument class during loading into JVM using service API provided in *java.lang.instrument* package. In our product we're using this type of weaving and we have to provide additional library which is responsible for weaving our code into existing actor's code.

For more information about AspectJ weaving types, please see Denis Zhdanov's blog post [9].

1.3 SBT

SBT (Simple Build Tool) is a project building tool that is commonly used to build Scala projects. It's written in Scala language and provides many great features, such as:

- incremental compilation - the tool scans the project sources and detects which files have been changed and recompiles only them - the compilation time is therefore greatly reduced,
- small configuration needed for simple projects,
- dependency management - manages the project's dependencies using Apache Ivy 2,
- packages applications into jar/war files,
- provides plugin architecture which enables users to write plugins which enable functionality that the tool itself do not have,
- provides ways to enable automatic code generation.

The build configuration files are written in Scala language. This is a great feature because it allows user to write some code which will be executed during tool setup or some build task.

More information about the SBT features can be found in SBT documentation [10].

1.3.1 Tasks

The SBT Documentation [11] defines tasks as values “that have to be recomputed each time, potentially with side effects”. Tasks are used, for example, to compile the project, to run unit tests, to run main class, etc.

Tasks are mostly used for providing some additional functionality - for example generating documentation or generating code.

1.3.2 Plugins

The plugins provide a way to define custom tasks that can be reused by other SBT users. This approach enables the community to extend the SBT functionality and to provide user-friendly way for some of the needed build steps required when using using some library or framework.

For example, Play framework (MVC web application framework) uses a plugin to provide routes generation during the compilation. This enables users to write a simple routes definition file and not bother to create lots of classes with mostly boilerplate code.

In Figure 2 we can see one of the usages of plugins - requiring custom task to be done before compilation. The custom task is shared to SBT users via plugins.

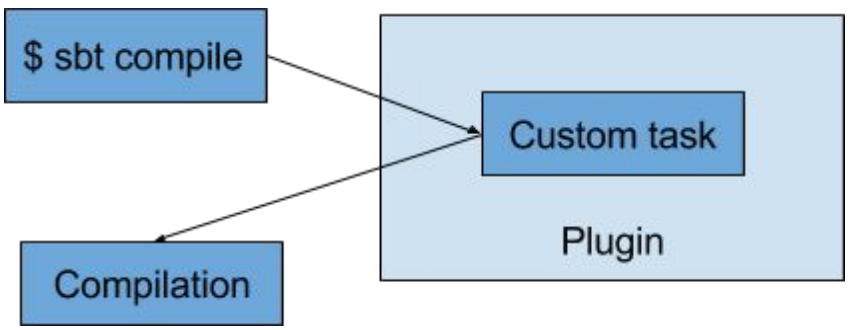

Figure 2. Usage of plugin to provide custom task to the SBT community

More information about SBT can be found in official SBT documentation [12].

2 Project structure

2.1 Library architecture

We split the library into three main parts to help us maintain the Project. In Figure 3 we can see the overall product structure.

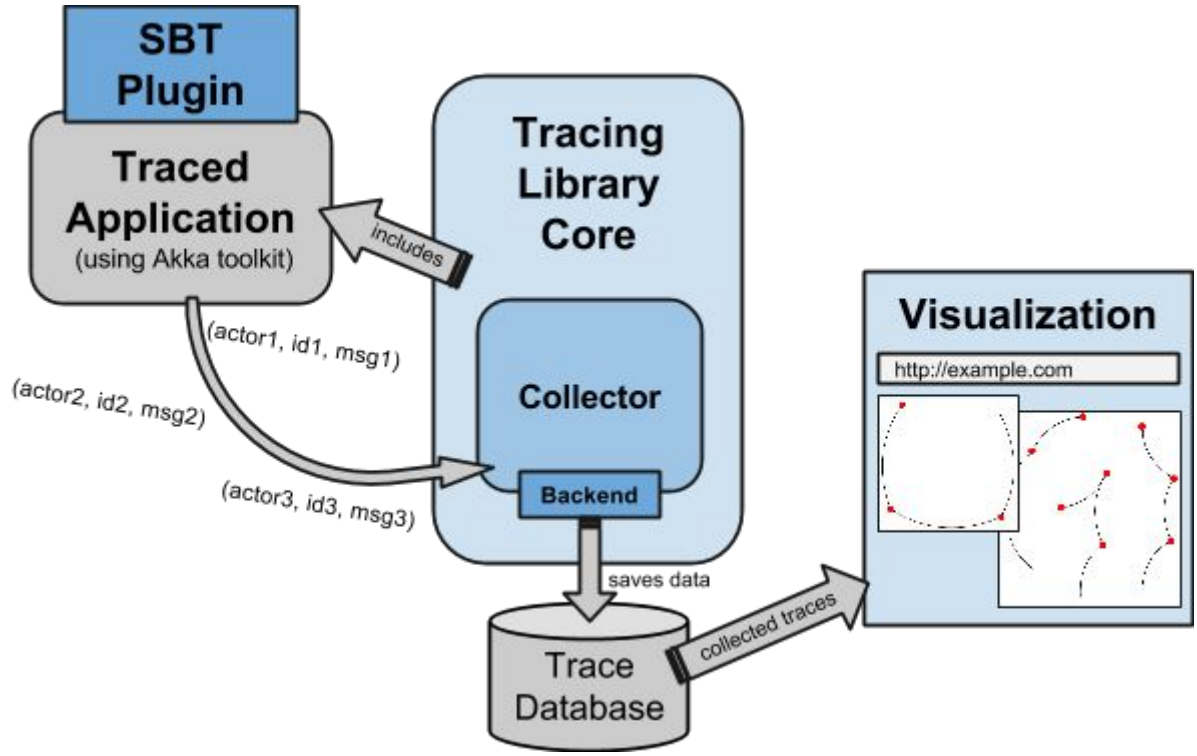

Figure 3. The overall product structure (the blue elements are parts of the Project)

2.2 Library core

The core of our library consists of classes that provide persistence of the message that is passed between actors, and additional facilities to connect messages into traces.

The main functionality that the library's core provides are:

- Collector actor which provides the persistence of the messages captured by our library.
- `TracedActor` trait which must be mixed into the actors which the user wants to trace.
- `MessageWrapper` class that wraps the message along with its UUID.
- Database access classes which provide the proper methods and database mappings used by different parts of the library.

The library's core is split into 3 main parts:

- **Tracing tools** - utility classes used by aspect (see Sections 2.2.1 and 2.3.3).
- **Collector** - actors which is created during aspect's initialization (see Sections 2.2.2 and 2.3.3)
- **Database** - database access classes used by whole library (see Section 2.2.3).

In Figure 4 we can see the class diagram of the library core.

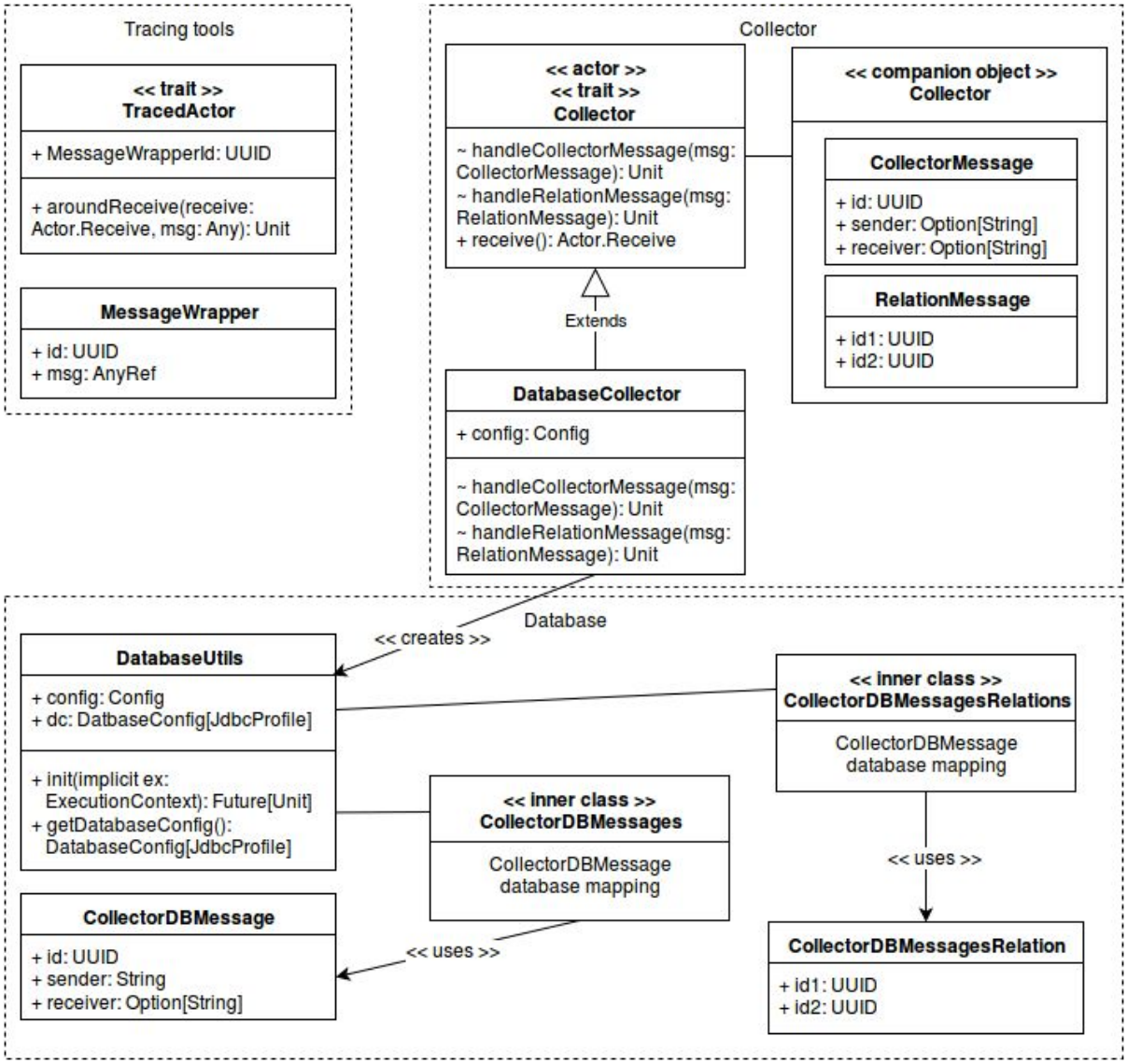

Figure 4. Class diagram in core

2.2.1 Tracing tools

The tracing tools are 2 classes: `TracedActor` trait which user must mix into the actors that he wants to trace and `MessageWrapper` class which adds the id of the message.

These classes are used by aspect which is generated by plugin (see Section 2.3.3.1 for more information). The `TracedActor` trait is used to identify the actors which should be traced by aspect's pointcut and the `MessageWrapper` provides identifier to the message which is later used to connect messages into traces.

2.2.2 Collector

Collector provides actor for the remote actor system started by aspect. The collector has 2 main classes:

- **`Collector`** - a trait that provides an interface for all possible implementations (backends) - the collector itself can be collected in different ways - it can persist messages in database, in file or in any other way we desire. The `Collector` trait has its companion object which defines the messages that can be passed to it:
  - **`CollectorMessage`** - this message contains information about either receiving or sending the message (depending on the fields' contents).
  - **`RelationMessage`** - this message contains information that message with `id1` happened in trace before message with `id2`.
- **`DatabaseCollector`** - implementation of `Collector` that uses user-defined database connection (user defines it in configuration file) to persist the messages. This implementation uses the database part of the core (see Section 2.2.3).

2.2.3 Database

Database provides the persistence of the messages in most relational database systems. It uses Slick v. 3.1.1 to provide the database connectivity and any database supported by Slick can be used by user. For more information about supported databases, please see Slick documentation [13].

The package consists of 3 parts:

- **`DatabaseUtils`** - a class that provides initalization and connection to the database for various library parts: collector, visualization and plugin.
- **Database class representations** - classes that represents rows in the database: `CollectorDBMessage` and `CollectorDBMessagesRelation`.
- **Database mappings** - inner classes of DatabaseUtils class that provides inner Slick mappings for the classes above. For more information about Slick, plese see the official Slick documentation [14].

The database that we use is really simple. It contains the following tables:

- **`messages`** - this table contains the messages captured by the library. It has the following content:
  - **`id`** - the identifier of the message (UUID),

    - **`sender`** - the canonical name of the actor's class that sent the message (nullable),
    - **`receiver`** - the canonical name of the actor's class that received the message (nullable).
- **`relation`** - this table contains the succession relation which provides information which message happened after which message. It has the following content:
    - **`id1`** - the identifier of the predecessor message (UUID),
    - **`id2`** - the identifier of the successor message (UUID).

In Figure 5 we can see the ERD diagram of the database.

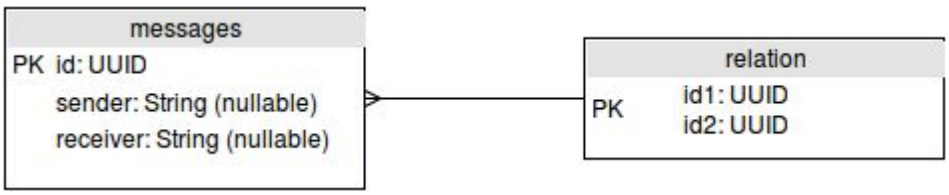

Figure 5. The ERD diagram of the used database

2.3 SBT Plugin

The SBT plugin is an essential part of the library. It provides the following functionality in our product:

- parsing configuration file to get information about actors that user want to trace with our library,
- generating an AspectJ aspect using the collected information,
- generating AspectJ weaving configuration,
- adding library core as user's project dependency with the necessary dependency resolver,
- adding AspectJ dependencies and weaving options to the user's project,
- providing database utility tools which enables user to create needed tables in the database and clean database of collected data.

The functionalities mentioned above are essential parts of our solution as they actually cause that the usage of the library is easy and intuitive.

The plugin itself is divided in four parts:

- **the "main" object** - it is something like the main function in C/C++ programs. It actually connects other parts together and provides the settings being added to the user's build definition file,
- **the configuration reader and parser** - gets information about packages and actors in these packages that user wants to trace,

- **aspects template and generator** - based on the information got from the configuration generates aspects which are responsible for instrumenting the code of our library which persists the messages in the database,
- **database utilities** - utility tasks for creating necessary tables and cleaning data stored in the database.

In Figure 6 we can see the class diagram in the plugin.

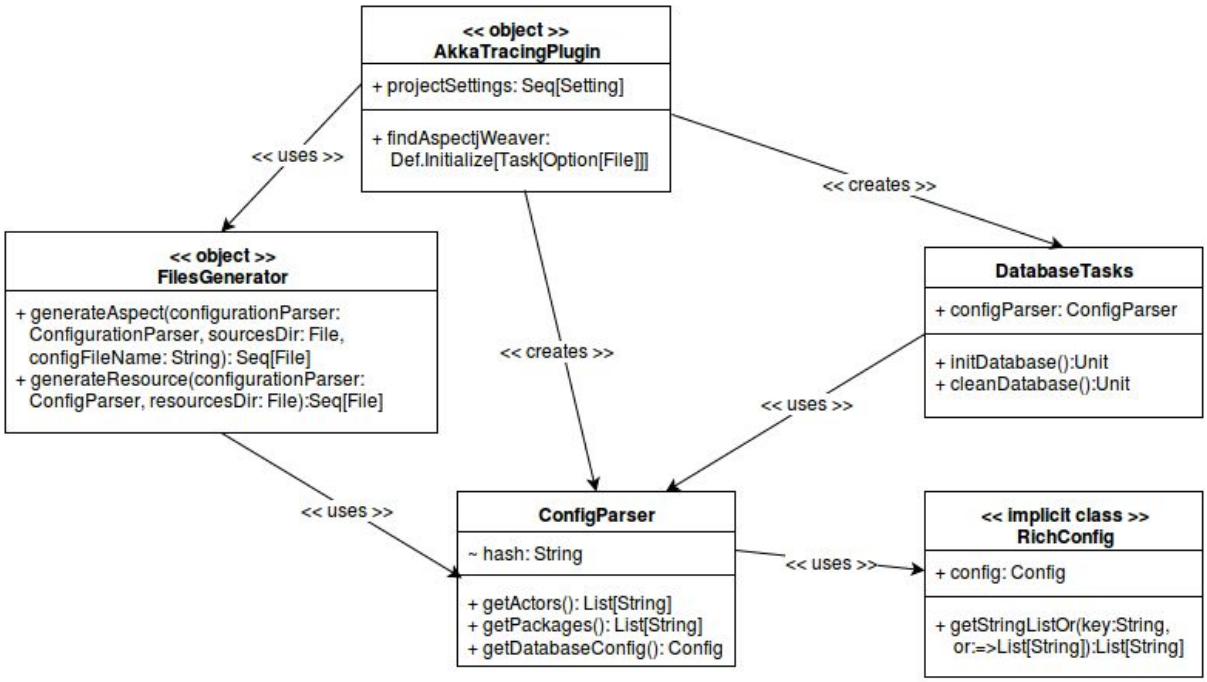

Figure 6. Class diagram in plugin

2.3.1 Plugin “main” object

The plugin main object, `AkkaTracingPlugin`, provides settings that will be added to user’s build definition file. It is responsible for the following functionality:

- providing **aspect generation** by providing SBT’s source and resource generators (see Section 2.3.3 for more details),
- providing **AspectJ dependencies** with **AspectJ weaving JVM option** needed to run the generated aspects,
- providing **database utilities tasks**: `initDatabase` and `cleanDatabase`. The initialization task is also declared as dependency in compilation to prevent SQL errors during running of the actor system. The collector sometimes does not have enough time to run these commands before insertions to the database and it caused loss of information - that’s why we moved initializing to be part of the compilation process (see Section 2.3.4 for more details).

To provide the functionality mentioned above, the main object uses also configuration parser to read the user's configuration. For more details about configuration parser, see Section 2.3.2.

The plugin uses standard SBT features - it defines proper settings and tasks and overrides `projectSettings` variable to include them into user's build definition file.

Due to the presence of source and resource generators in the plugin, it was necessary to add `JvmPlugin` dependency because `JvmPlugin` provides and overrides the source and resource directories (for more information about this see StackOverflow thread [15]).

2.3.2 The configuration reader and parser

The configuration reader and parser is encapsulated in the `ConfigParser` class. It provides methods and fields for getting the following values:

- packages containing actors to trace,
- actors that user wants to trace in each package (supplied explicitly by the user or, if not specified, it is assumed that any found actor with mixed trait TracedActor will be traced by the library),
- hash of the configuration file (calculated with SHA-512 algorithm),
- database connection configuration used by database utilities.

The class is used in all other parts in plugin. One of the most important thing is the hash calculation. It prevents the plugin to generate the aspect and weaving config if they were already generated using the same configuration. It's useful because it saves time during compilation.

Due to the chosen configuration format, the parser and configuration reader used was TypeSafe's Config library. The library, while providing really great interface, was lacking Scala's syntactic sugar in error handling. If an error occured, it was reported by throwing an exception. While it's common in Java code, in Scala we would like to handle errors in different ways. That's why the implicit class `RichConfig` was written. It provides an additional method which enables us to provide the default value while trying to get list of strings.

2.3.3 Aspects generator

The main functionality of the plugin is generating an AspectJ aspect which instruments the library code.

The generation consists of 2 parts: generating the aspect itself and the weaving configuration file.

The class that provides the generation functionality is named `FilesGenerator`. It needs the `ConfigParser` instance to provide the necessary configuration - list of traced actors and configuration hash.

The class takes the aspect template from resources and replaces the needed values with the ones read from configuration. This is possible due to the simple construction of the aspect itself.

The aspect is saved in the managed sources directory which allows user to not have the additional source in his source directory and SBT to compile the file anyway. This solution increases the user-friendly aspect of our library. Similarly, the weaving configuration is saved to the managed resources directory.

Aspect

The aspect provides following features:

- The `aroundReceivePointcut` which points to the `TracedActor`'s `receive` method. It is catching any invocation of this method which allows us to instrument our code.
- The `aspectAroundReceive` advice which unwraps the message from our message wrapper and passes it to the `receive` method in actors and sends the messages to the database collector which persists it in the database. If the message is not wrapped in our wrapper, it would be passed to the actor without saving it to database.
- The `withinUnreliable` pointcut which points to the `tell` (`!`) method in actors. It is catching sending the messages between actors.
- The `aspectA` advice which wraps the message being send and also connect the messages using the `correlationId` from the `TracedActor` trait into traces. The id itself is a random UUID.

The aspect also starts a new remote actor system which contains the collector actor from the core library.

2.3.4 Database utilities

The plugin provides also utility tasks to manage the database used in tracing. The responsible class, `DatabaseTasks`, has two methods:

- **`initDatabase`** which creates two tables: messages and relation that are needed by the library to persist the information into the database,
- **`cleanDatabase`** which cleans the tables mentioned above.

The class uses Slick library to run the proper SQL commands and database configuration read from the configuration file to provide the proper database connection.

2.4 Visualization tool

The purpose of the visualization tool was to confirm that the library is working properly. The tool is more like a by-product than part of our Project. The main objective was to provide visual confirmation that the information saved to the database is correct and that proper traces can be retrieved from it.

The tool is a Play! 2.4 application. The tool itself is very simple and provides the graphical visualization of the collected traces by displaying them as a directed graph. It does not have any special features like filtering traces due to being a side product.

For more information about Play! framework please see the official Play! framework documentation [16].

The “backend” has very simple task - it reads information from the database and passes it to the “frontend” graph converter.

The graph converter was written in Javascript. Our tool uses sigma.js library to render the graph in the user interface. Due to incompatible graph representations, the graph converter needs to perform a DFS search in the data got from the backend to get the graph representation needed by sigma.js.

The user interface was created using the Bootstrap library. It allowed us to focus on the functionality we wanted to deliver and not about styling the application.

In Figure 7 we can see the overall tool structure. In Figure 8 we can see the visualization of the very simple message passing scenario. In Figure 9 we can see the visualization of a more sophisticated scenario.

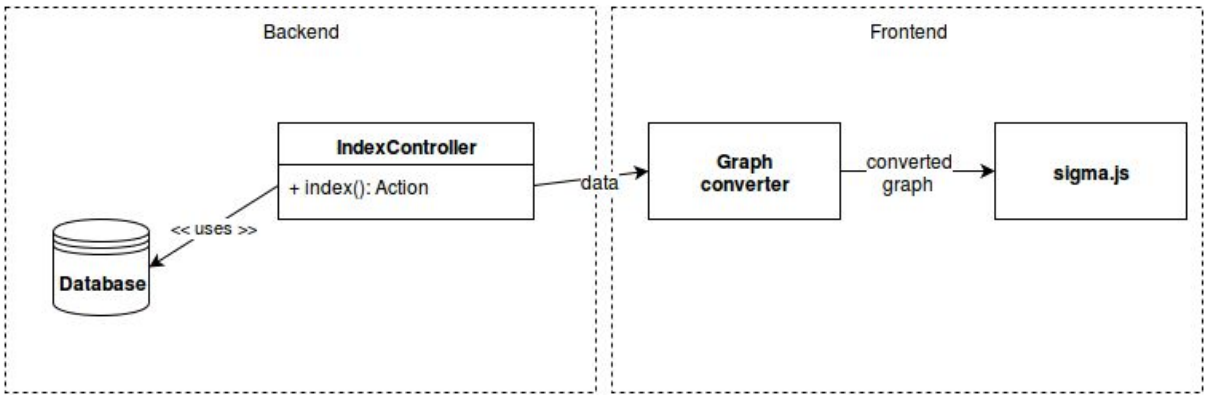

Figure 7. Visualization tool structure

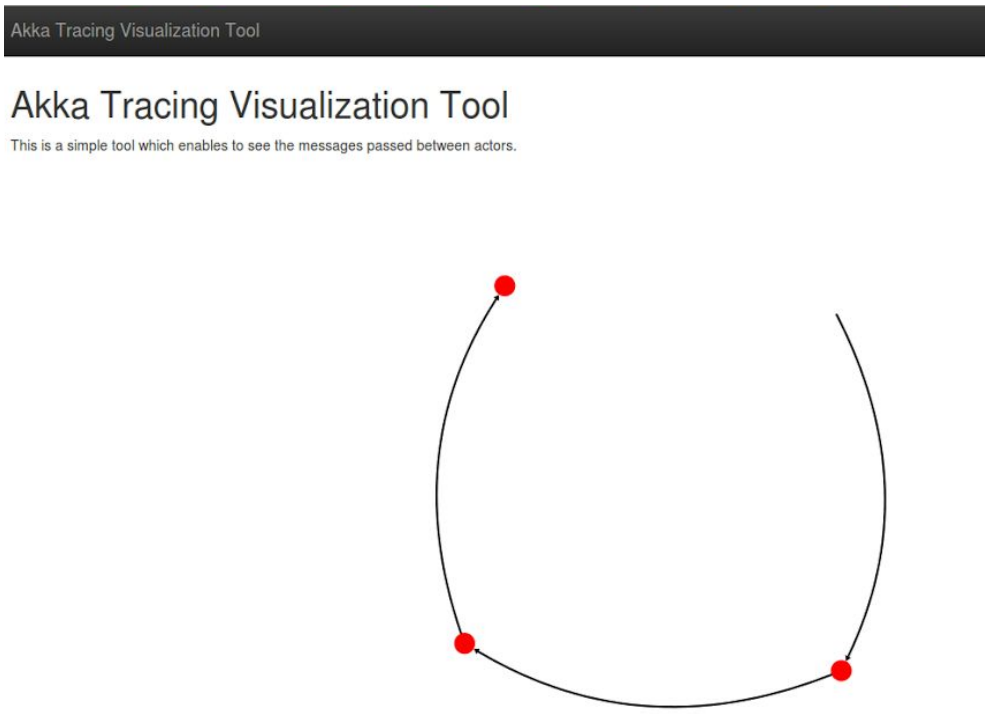

Figure 8. Simple scenario visualization using our visualization tool

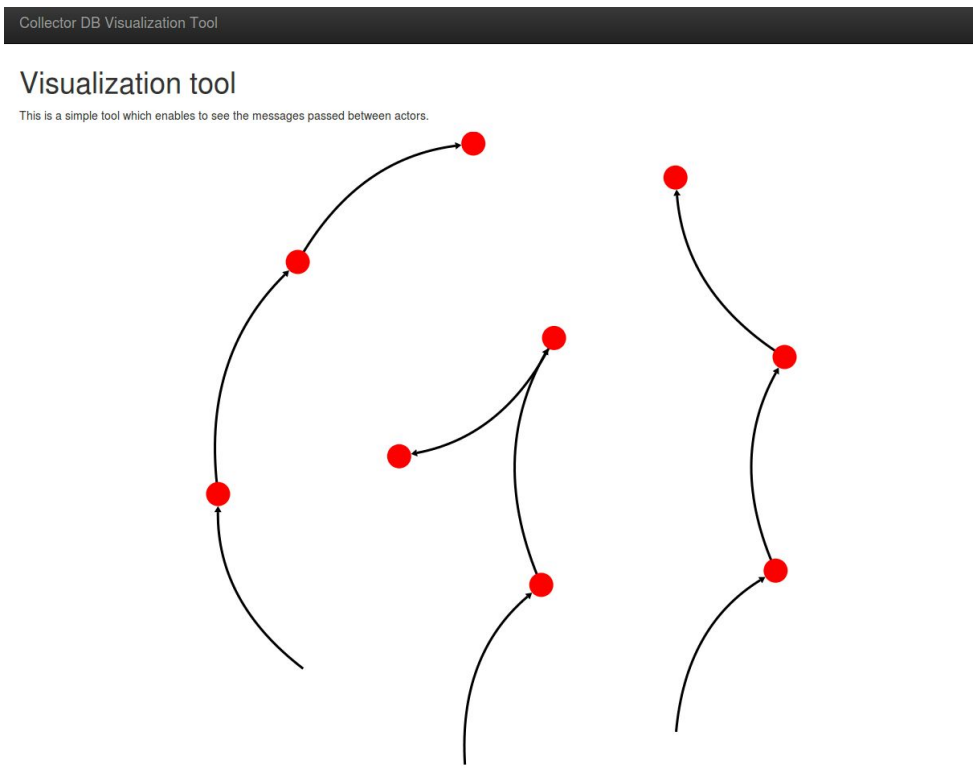

Figure 9. More sophisticated scenario visualized using our visualization tool

3 Tests

In the initial phase of this Project, we were mainly focused on creating prototypes and fast delivering it to client to asses result. It caused that early versions was lacking in tests and automatic verification of output. This was also connected with product requirements which weren't very strict and precisely determinate at the beginning of our work: we had to recognize the best solution for client needs. Hence, introducing test-driven development into our workflow was nearly impossible and covering the whole codebase with automated tests was very hard because of constant changes.

Although, we saw benefits of having tests in our Project: an assurance that everything works, e.g. we didn't introduced any bugs that cause a regression, and client can check consistent of created software with his requirements. In the later phase we decided to introduce CI system, Travis, which after every commit tried to compile whole project and reported it on code repository pages. It increased reliability of our process and every time we knew that we were developing further working version of our product.

As mentioned in Development Process Documentation in Section 4, some of checks, connected mostly with non-functional requirements, were impossible to automate. We chose to focus on very simple tests, e.g. one message passed between two actors. We didn't check sophisticated examples but straightforward scenarios because it was achievable in our case. We also found "smoke tests" very useful: it allowed us to check whether updated version of dependencies didn't introduce regression and make sure that everything work properly.

Below we present a list of tests in our Project:

- Do `initDatabase`/`cleanDatabase` methods in `DatabaseTasks` work? - The test checks whether database is properly initialized and cleaned after invocation of these methods.
- Is configuration properly parsed? - Simple check whether correct values are returned from `ConfigParser` class.
- Does `FilesGenerator` produce correct output file? - The test that makes sure whether file content is correct.
- Does `FilesGenerator` generate aspect when necessary? - The tests that check whether new file is generated when configuration changed.
- Does implicit class `RichConfig` work properly? - Simple check that makes sure implicit class works as expected.
- Does Collector put proper data into database after receiving messages? - The test that checks amount of rows after receiving specified messages.
- Simple integration test - a script that run application and check whether output: database and generated files are correct.

4 Information for developers

4.1 Prerequisites

There are several things that needs to be installed before you can start developing and contributing to Akka Tracing Tool.

First of all, you need to have Java 8 JDK installed on your PC. The preferred implementation is the Oracle one, but it should be fine on others (though this was not tested). Java 8 JDK (Oracle implementation) can be downloaded from the Oracle's website [17].

Also, the Scala language should be installed in your system. We used two versions: 2.11.7 in most of our library, but the SBT plugin required version 2.10.5. Because of the plugin's dependency on the core, it needed to be cross-compiled in two versions (2.10.5 and 2.11.7). Scala can be downloaded from its official website [18].

The next important tool is git version control system. It is used in any part of the Project. Git can be downloaded from its official website [19].

The last needed tool is the SBT building tool. It's used to compile all the library's code as well as run unit tests. We used its version 0.13.9. SBT tool can be downloaded from its official website [20].

Any needed libraries that *Akka Tracing Tool* uses will be downloaded by SBT during the compilation or the project's import to your favourite IDE.

4.2 Importing project into IntelliJ IDEA

After installing all of the tools mentioned above, you can import the project you want to edit to your favourite IDE. Below we present the description of the process of importing the library's core to IntelliJ IDEA.

Before we start, you have to make sure that Scala plugin in IntelliJ IDEA is installed. It's available in public JetBrains repository and you can download it inside IDE. In order to do that, please open in settings "Plugin" section. Then, please press "Browse repositories…". You should see a window like in Figure 9. As a final step, press the button to start installation of the plugin.

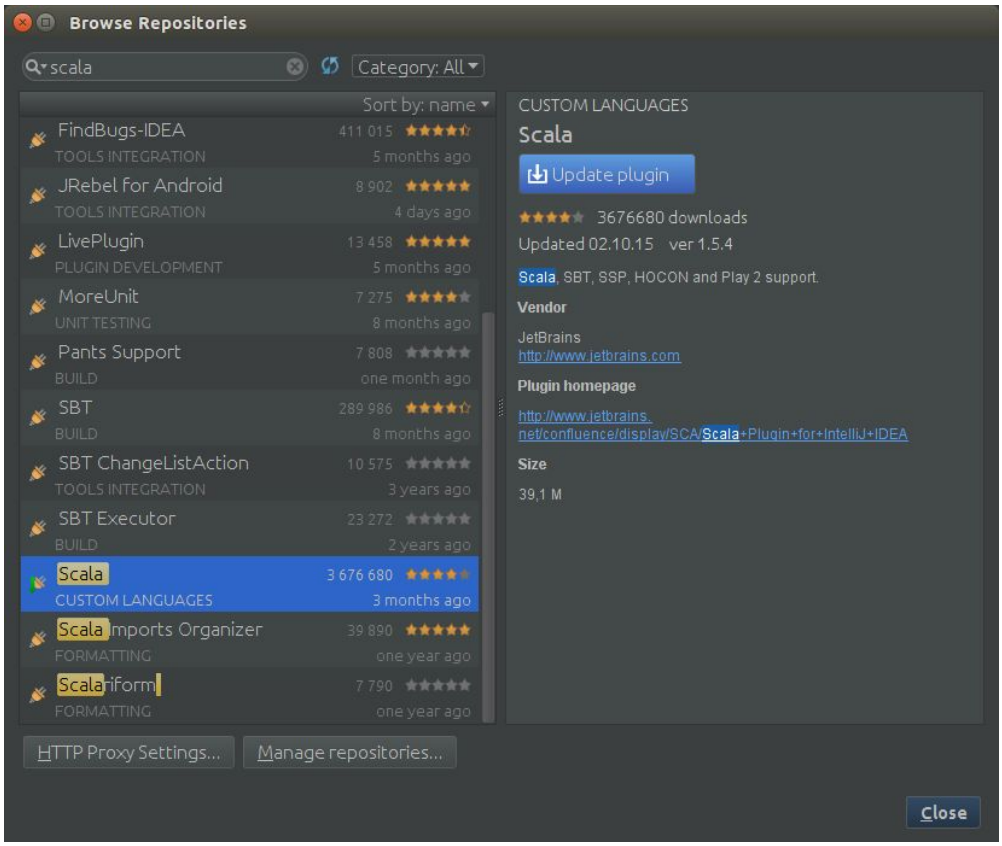

Figure 9. "Browse Repositories" window in plugins settings

After that we can open the SBT project.

Right after IntelliJ IDEA is initialized, you should see a window similar to one in Figure 10:

Figure 10. Welcome screen in IntelliJ IDEA

In the window visible above you should click “Open” link. After that, new window - shown in Figure 11 - should pop up. You have to find project cloned from GitHub in your filesystem, mark it and press the “OK” button.

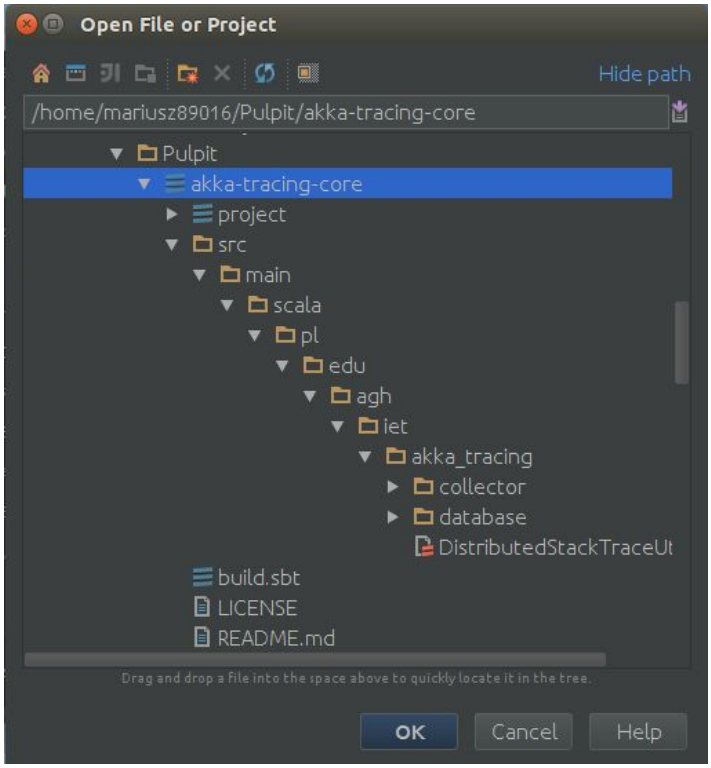

Figure 11. Pop-up “Open file or project” window

If you have done all of the previous steps, you should proceed to the new window, shown in Figure 12. Here you can modify different settings. The most important are SBT project and project SDK. Also, it’s convenient to turn on “auto-import” capability.

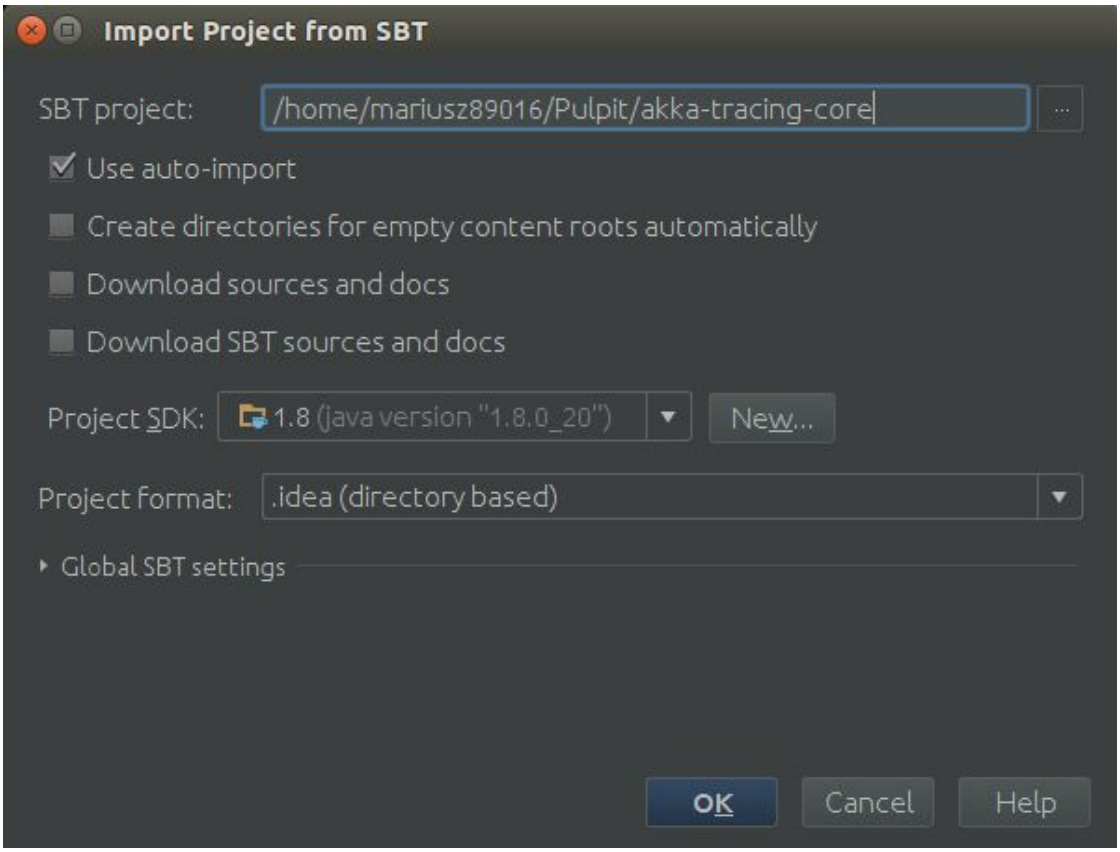

Figure 12. “Import project from SBT” window

When you press “OK” button, you should see small pop-up window that informs you about progress building a IntelliJ IDEA project. It’s visible in Figure 13 below.

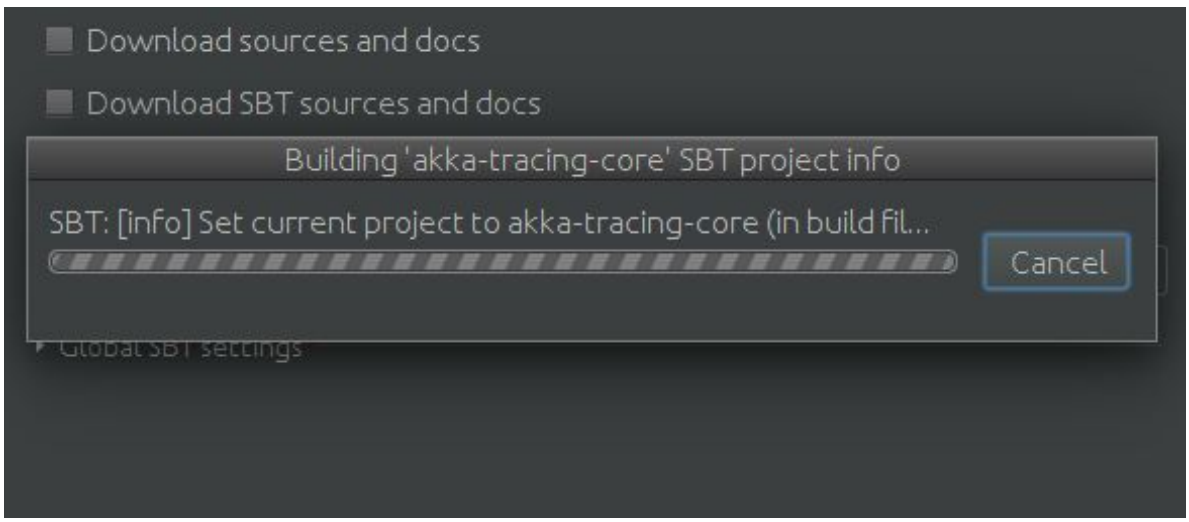

Figure 13. Process of building IntelliJ IDEA project

When the previous step is done, the project is correctly imported into IntelliJ IDEA. In Figure 14 there is a screenshot from the IDE after importing project. Now, you can start developing Akka Tracing Tool.

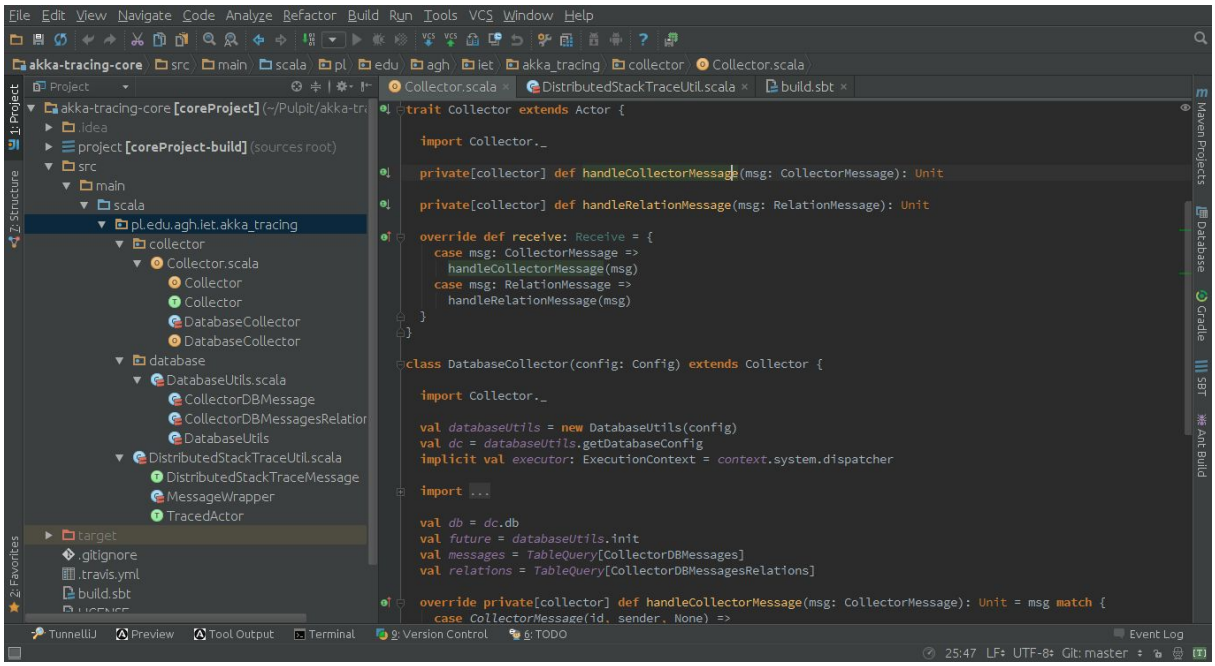

Figure 14. Screenshot from opened “akka-tracing-core” SBT project in IntelliJ IDEA

4.3 Contributing to Akka Tracing Tool

If you want to send us some bug fix or contribute to the library’s development, please fork the necessary repository on GitHub and create a pull request. The process is thoroughly described in GitHub’s tutorial [21].

References

[1] Akka documentation, “Actors” section. Available at: http://doc.akka.io/docs/akka/2.3.9/scala/actors.html#actors-scala [Online; accessed 31.12.2015]

[2] Akka documentation, “Actors and shared mutable state” section. Available at: http://doc.akka.io/docs/akka/snapshot/general/jmm.html#Actors_and_shared_mutable_state [Online; accessed 31.12.2015]

[3] Alvin Alexander, “An Akka actors 'ask' example - ask, future, await, timeout, duration, and all that” blog post. Available at: http://alvinalexander.com/scala/scala-akka-actors-ask-examples-future-await-timeout-result [Online; accessed 31.12.2015]

[4] J. Waldo, G. Wyant, A. Wollrath, S. Kendall, “A note on Distributed computing”, November 1994. Available at: http://doc.akka.io/docs/misc/smli_tr-94-29.pdf [Online; accessed 31.12.2015]

[5] Akka documentation, “Location transparency” section. Available at: http://doc.akka.io/docs/akka/snapshot/general/remoting.html#remoting [Online; accessed 31.12.2015]

[6] Akka documentation. Available at: http://akka.io/docs [Online; accessed 31.12.2015]

[7] Wikipedia article “Aspect-oriented programming”. Available at: https://en.wikipedia.org/wiki/Aspect-oriented_programming [Online; accessed 31.12.2015]

[8] Wikipedia article “AspectJ”. Available at: https://en.wikipedia.org/wiki/AspectJ [Online; accessed 31.12.2015]

[9] Denis Zhdanov, “Weaving with AspectJ” blog post. Available at: http://denis-zhdanov.blogspot.com/2009/08/weaving-with-aspectj.html [Online; accessed 31.12.2015]

[10] SBT documentation, “Features of SBT” section. Available at: http://www.scala-sbt.org/0.13/docs/index.html#Features+of+sbt [Online; accessed 31.12.2015]

[11] SBT documentation, “.sbt build definition” section. Available at: http://www.scala-sbt.org/0.13/tutorial/Basic-Def.html [Online; accessed 31.12.2015]

[12] SBT documentation. Available at: http://www.scala-sbt.org/documentation.html [Online; accessed 31.12.2015]

[13] Slick documentation, “Supported databases” section. Available at: http://slick.typesafe.com/doc/3.1.1/supported-databases.html [Online; accessed 31.12.2015]

[14] Slick documentation. Available at: http://slick.typesafe.com/doc/3.1.1/ [Online; accessed 31.12.2015]

[15] StackOverflow thread: “How to generate sources in an sbt plugin?”. Available at: http://stackoverflow.com/questions/24724406/how-to-generate-sources-in-an-sbt-plugin [Online; accessed 31.12.2015]

[16] Play documentation. Available at: https://playframework.com/documentation/2.4.x/Home [Online; accessed 31.12.2015]

[17] Oracle website, “Java SE Development Kit 8 Downloads” section. Available at: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html [Online; accessed 03.01.2016]
[18] The Scala Programming Language website, “Downloads” section. Available at: http://www.scala-lang.org/download/all.html [Online; accessed 03.01.2016]
[19] Git website, “Downloads” section. Available at: https://git-scm.com/downloads [Online; accessed 03.01.2016]
[20] SBT website, “Downloads” section. Available at: http://www.scala-sbt.org/download.html [Online; accessed 03.01.2016]
[21] GitHub Help website, “Fork a repo” Bootcamp. Available at: https://help.github.com/articles/fork-a-repo [Online; accessed 03.01.2016]