Programowanie reaktywne w języku Scala

Wprowadzenie do języka Scala oraz do Akka Actors

Strona domowa: http://www.scala-lang.org

Coursera Functional Programming Principles in Scala

Środowisko potrzebne do wykonywania ćwiczeń:

- JDK 1.6 lub 1.7
- sbt - build tool
- Scala IDE for Eclipse UWAGA: należy pobrać wersję dla Scala 2.10.4 .

Zadanie

- Pobrać projekt eclipse potrzebny do wykonania ćwiczenia. Rozpakować i otworzyć go w Scala IDE (File->Import-general->Existing Project into Workspace)
- Wykonać zadania w arkuszach od tutorial.cs do tutorial4.cs. Są to interaktywne arkusze Scali czyli Scala Worksheet
- Zapoznać się i uruchomić przykłady aktorów znajdujących się w src/main/scala. Wymagana konfiguracja.

Bartosz Baliś, balis at agh edu pl
Maciej Malawski, malawski at agh edu pl
Katarzyna Rycerz, kzajac at agh edu pl

Programowanie reaktywne w jezyku Scala

Projektowanie systemu aktorów

Przykłady

- Pobrać projekt eclipse, rozpakować go i otworzyć w Scala IDE (File->Import-general->Existing Project into Workspace)
- Przyklad BuddyChat

  Uruchomienie przykładu:

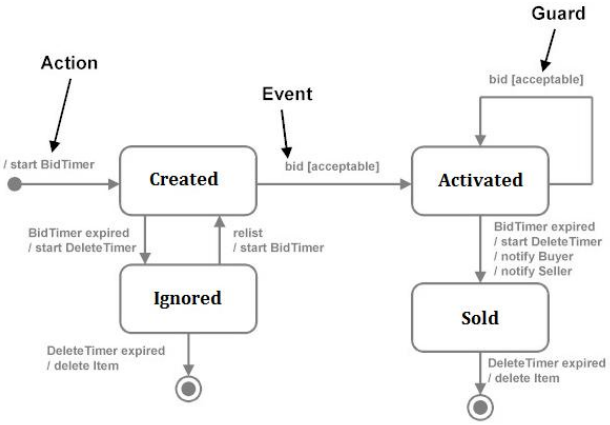
```
$ git clone https://github.com/sdanzig/buddychat
$ cd buddychat
==> Prosze podmienić plik build.sbt
$ sbt run
```

- Dla zainteresowanych: konwertowanie projektów sbt do eclipse
- Akka FSM

Zadanie - system aukcyjny

Zadanie polega na zaimplementowaniu fragmentu systemu aukcyjnego z nastepujacymi aktorami:

1. **Auction**: aktor reprezentujący aukcje, ktory ma dzialac wedlug nastepującej maszyny stanow:



2. **Buyer**: aktor, ktory licytuje w istniejacych aukcjach.

Zadania do wykonania:

1. (20 pkt) Prosze zaimplementować i uruchomic uproszczony system, w ktorym na poczatku tworzona jest pewna liczbe aukcji (aktorow **Auction**) oraz kilku aktorow**Buyer**, ktorzy licytuja w istniejacych aukcjach.
   - Prosze zaprojektować wiadomości jako Case Class'y
   - Proszę zaprojektować niezbędny wewnetrzny stan aktorow.
   - Prosze sprawdzac czy licytacja jest poprawna, tj. czy oferta w licytacji przebija aktualna cene.
   - Aktorzy **Buyer** moga dostac liste referencji (ActorRef) do aktorow aukcji przez konstruktor.
2. (10 pkt) Prosze zaimplementować i uruchomic pelna maszyne stanow.
   - Timery prosze zrealizowac przy pomocy Schedulera: aktor powinien zaplanowac wyslanie wiadomosci do samego siebie oznaczajacej upluniecie okreslonego terminu (BidTimer: konca licytacji, DeleteTimer: usuniecia licytacji z systemu).
3. (10 pkt)
   - Prosze wykorzystac Akka FSM do implementacji maszyny stanow w Aktorze **Auction**.

Bartosz Baliś, balis at agh edu pl
Maciej Malawski, malawski at agh edu pl
Katarzyna Rycerz, kzajac at agh edu pl

Programowanie reaktywne w jezyku Scala

Testowanie aktorów

- Do testowania aktorów wykorzystujemy moduł akka-testkit . W przykładzie rozszerzana jest klasa TestKit inicjalizowana systemem aktorów do testowania.
- Akka TestKit tworzy wewnętrznego testującego aktora (o nawie testActor), który służy do komunikacji z testowanymi aktorami
- Odpowiedzi od testowanych aktorów są kolejkowane przez testującego aktora i mogą być badane przy użyciu metod takich jak *expectMsg* .
- Uzycie traitu *ImplicitSender* powoduje, że aktor testujący jest domyślnie używany jako ten, który wysyła wiadomości do testowanych aktorów.

Uzycie akka testkit w ScalaTest

- Jako narzędzia do testowania używamy ScalaTest
- ScalaTest oferuje różne style testowania czyli sposoby deklarowania testów. W przykładzie wykorzystano naturalny styl WordSpec (trait *WordSpecLike*)
- Trait *BeforeAndAfterAll* zapewnia funkcje do wykonania przed i po zestawie testów. W przykładzie pokazano, jak wykorzystać funkcje *afterAll* do zamknięcia systemu aktorów po zakonczeniu testów.

Przykład prostego testu

- Przykład testuje aktora Toggle z poprzedniego przykladu z laboratorium 2
- W zestawie zawarte są trzy testy w stylu *WordSpec*

```
class ToggleSpec extends TestKit(ActorSystem("ToggleSpec"))
 with WordSpecLike with BeforeAndAfterAll with ImplicitSender {

 override def afterAll(): Unit = {
   system.shutdown()
 }

 "A Toggle" must {

   "start in a happy mood" in {
     val toggle = system.actorOf(Props[Toggle])

     toggle ! "How are you?"
     expectMsg("happy")
   }

   "change its mood" in {
     val toggle = system.actorOf(Props[Toggle])
     for (i <- 1 to 5){
       toggle ! "How are you?"
       expectMsg("happy")
       toggle ! "How are you?"
       expectMsg("sad")
     }
   }

   "finish when done" in{
     val toggle = system.actorOf(Props[Toggle])
     toggle ! "Done"
     expectMsg("Done")
   }
 }
}
```

Jak uruchomić testy ?

- Pobrać projekt potrzebny do wykonania ćwiczenia.
- Przykład testu znajduje sie w pliku src/test/scala/myActorTest/ToggleSpec.scala
- Projekt można wykonać pod sbt
    - komenda run powoduje uruchomienie głównego programu znajdujacego sie w src/main/scala
    - komenda test uruchamia testowanie za pomocą ScalaTest
    - Pełna instrukcja jak używać ScalaTest w SBT
    - konfiguracja sbt zapisana jest w pliku build.sbt projektu (wszystkie potrzebne zależności)
- Projekt można też otworzyć w Scala IDE (File->Import-general->Existing Project into Workspace)
    - Aby uruchomić przykład aktorów znajdujących się w src/main/scala należy wybrać *Run as Scala Application*
    - Aby uruchomic test aktora Toggle znajdujacy się w src/test/scala należy wybrać *Run as -> ScalaTest (File)*
    - Aby wykonać ten krok na własnym Scala IDE wymagana jest instalacja pluginu ScalaTest pod Eclipse . Plugin ten jest już zainstalowany na obrazie ICSR na PCoIP. Wskazówki:
        - wybrać z menu Scala IDE Help -> Install New Software
        - wpisac w Work with *http://download.scala-ide.org/sdk/e38/scala210/stable/site*
        - po pojawieniu sie pakietow zaznaczyc ScalaTest i kliknać Next

Ważne linki

- Testowanie aktorów - dokumentacja akka

Zadanie

1. (20 pkt) Prosze dodac do systemu aukcyjnego aktorow **Seller** i **AuctionSearch**
    - **Seller** wystawia kilka aukcji o roznych tytulach (przykladowo "Audi A6 diesel manual"). Kazdemu aktorowi Seller mozna np. przekazywac przez konstruktor liste tytulow aukcji, ktore ma wystawic.
    - **AuctionSearch** to aktor, w ktorym mozna wyszukiwac aukcje. Zapytanie wyszukiwania moze skladac sie z jednego slowa i jesli slowo to wystepuje w tytule aukcji, to aukcja pasuje do zapytania.
    - Kazda aukcja musi rejestrowac sie w **AuctionSearch**. Prosze wykorzystac mechanizm Actor Selection do wyszukiwania aktora **AuctionSearch**.
    - Aktorzy **Buyer** powinni korzystac z wyszukiwania przez **AuctionSearch** aby uzyskac referencje do iteresujacych ich aukcji.
2. (10 pkt) Wykorzystujac Akka TestKit Prosze napisac testy dla aktora Auction.
3. (10 pkt) Prosze rozszerzyć funkcjonalność oraz stworzyc test calego systemu aukcyjnego:
    - Proszę rozszerzyć Aktora **Buyer** o nowe zachowanie (stan), w którym zapisuje się na notyfikacje, gdy ktoś przebije jego aktualna ofertę i samemu przebija ofertę, jeśli cena nie przekroczyła jego kwoty maksymalnej. Prosze rozszerzyć Aktora **Auction** o potrzebne wsparcie dla takiego zachowania.
    - Prosze stworzyć testowy zbiór danych okreslający sposób tworzenia i zachowanie systemu aktorow: (i) ile aukcji ma byc utworzonych (z podzialem na sprzedających) oraz ich tytuly, (ii) ilu ma byc kupujących, jakimi aukcjami maja sie interesować (slowo kluczowe) i jaka maksymalną kwotę licytować.
    - Proszę stworzyć system aktorów i wykonac testy przy pomocy Akka TestKit.

Bartosz Baliś, balis at agh edu pl
Maciej Malawski, malawski at agh edu pl
Katarzyna Rycerz, kzajac at agh edu pl

Programowanie reaktywne w języku Scala

Skalowanie systemu aktorów

- Do skalowania aktorów wykorzystujemy koncepcję routingu dostępnego w routerach Akka.
- Przykładowy projekt używający routera Akki
- Projekt można wykonać przy pomocy komendy `sbt run`.
- Paczka zawiera też wygenerowany projekt w eclipse (bez bibliotek). Biblioteki można ściągnąć komendą `sbt eclipse with-source=true`.

Zadanie

1. (20 pkt) Proszę dodać do systemu aukcyjnego aktora **MasterSearch**
   - **MasterSearch** zarządza wieloma aktorami **AuctionSearch** poprzez mechanizm routingu
   - Aktorzy **Seller** rejestrują aukcje poprzez **MasterSearch**
   - Aktorzy **Buyer** dokonują zapytań wyszukiwania również poprzez **MasterSearch**
   - (**Wariant z partycjonowaniem**) Zaimplementować wariant, w którym rejestracja aukcji odbywa sie u jednego aktora **AuctionSearch** poprzez logikę routera*RoundRobinRoutingLogic*, a wyszukiwanie odbywa się u wszystkich aktorów **AuctionSearch** (logika *BroadcastRoutingLogic*).
   - (**Wariant z replikacją**) Zaimplementować wariant, w którym rejestracja aukcji odbywa sie u wszystkich aktorów **AuctionSearch** poprzez logikę routera*BroadcastRoutingLogic* (redundacja danych), a wyszukiwanie u jednego aktora **AuctionSearch** (logika *RoundRobinRoutingLogic*).
2. (10 pkt) Wykonać testy wydajnościowe obydwu przypadków. Jak długo trwa pojedyncze wyszukiwanie w zależności od liczby zarejestrowanych aukcji oraz obciążenia systemu zapytaniami wyszukiwania? Jak długo trwa pojedyncze wyszukiwanie w zależności od różnej ilości aktorów *AuctionSearch*? Jaki jest średni czas odpowiedzi? W jakim czasie mieści się 75%, 90% zapytań? Wyniki przedstawić na wykresach.
3. (10 pkt) Do dynamicznego skalowania Akka udostępnia mechanizm Dynamically Resizable Pool, gdzie używa domyślnego resizera. Proszę dodać ten mechanizm do wersji z replikacją danych i zbadać jego wpływ na działanie systemu.

Bartosz Baliś, balis at agh edu pl
Maciej Malawski, malawski at agh edu pl
Katarzyna Rycerz, kzajac at agh edu pl

Programowanie reaktywne w języku Scala

Persystencja aktorów

- W celu zapisywania stanu aktora, używamy wzorca `Event Sourcing` zaimplementowanego w bibliotece Akka persistence.
- Przykładowy projekt używający persystencji Akki
- Projekt można wykonać przy pomocy komendy `sbt run`. Można też z niego wygenerować projekt w eclipse: `sbt eclipse`.
- Proszę sprawdzić zachowanie aktora przy kilkukrotnym uruchomieniu projektu. Co się zmienia, gdy wyślemy do niego komunikat `Snap`?
- Projekt wykorzystuje wbudowaną bazę danych LevelDB do zapisu dziennika zdarzeń. Pliki bazy tworzone są w katalogu `journal/`.
- Baza używa wbudowanej implementacji w Javie, co skonfigurowane jest w pliku `src/main/resources/application.conf`.
- Przykład użycia persystencji dla aktora wykorzystującego `context.become()` zamieszczony jest w pliku: PersistentToggle.scala.

Zadanie

1. (20 pkt) Proszę dodać persystencję dla aktora **Auction**
   - **Auction** zapisuje zdarzenia reprezentujące aktualny stan aukcji oraz zmiany kontekstu aktora (przejścia `become()`).
   - Uwaga: aktorów persystentnych nie można łączyć z użyciem FSM.
   - W dzienniku zdarzeń należy zapisać też czas trwania aukcji, a przy odtwarzaniu stanu aktora (komunikat `RecoveryCompleted`) uwzględnić ten czas przy ustalaniu nowego `BidTimer`.
2. (10 pkt) Przetestować scenariusz, w którym przed upływem końca aukcji aplikacja zostaje wyłączona (np. poprzez `system.shutdown()`). Po ponownym uruchomieniu aplikacji stan aktorów **Auction** powinien być odtworzony z dziennika, natomiast aktorzy **Seller**, **Buyer** oraz **AuctionSearch** mogą być zainicjalizowani od nowa. Kupujący powinni powrócić do licytowania zgodnie ze swoją strategią. Proszę sprawdzić, czy wielokrotne przerywanie i ponowne uruchamianie systemu aktorów umożliwia kontunuowanie licytacji.
3. (10 pkt) Proszę porównać wydajność różnych mechanizmów persystencji (np. LevelDB z opcją native = true/false, InMemory, itp., lista wtyczek)

Bartosz Baliś, balis at agh edu pl
Maciej Malawski, malawski at agh edu pl
Katarzyna Rycerz, kzajac at agh edu pl

Programowanie reaktywne w języku Scala

Obsługa awarii

- W celu zapewnienia obsługi awarii używamy wzorca nadzoru aktorów (*supervision*)
- Przykładowy projekt: Akka Supervision Tutorial używający aktorów do obliczania wyrażeń arytmetycznych
- Pobranie i uruchamianie projektu:

```
git clone https://github.com/typesafehub/activator-akka-supervision.git
cd activator-akka-supervision/
sbt eclipse
```

- Proszę sprawdzić zachowanie systemu aktorów przy kilkukrotnym uruchomieniu projektu. Jak obsługiwane są losowe błędy?
- Proszę sprawdzić co się stanie, gdy wprowadzimy wyrażenie zawierające dzielenie przez 0.
- Fault tolerance in akka
- Wiecej informacji o ask "?" i Futures

Komunikacja HTTP/REST

- Do zapewnienia komunikacji pomiędzy aktorami a systemami zewnętrznymi używamy biblioteki Spray.
- Szablon projektu serwera: spray-template
- Pobranie i uruchomienie przy użyciu wbudowanego serwera HTTP:

```
git clone git://github.com/spray/spray-template.git my-spray-project
cd my-spray-project
sbt
test
re-start
```

Serwer jest dostępny pod adresem http://localhost:8080

- Przykładowy projekt klienta: my-spray-client.zip
- Uruchamianie klienta:

```
cd my-spray-client/
sbt run
```

- Do działania z eclipse IDE potrzebna jest NOWA wersja IDE dla Scali w wersji 2.11.2

Zadanie

1. (20 pkt) Proszę rozszerzyć system aukcyjny o mechanizm publikowania informacji o przebiegu aukcji do zewnętrznego systemu **Auction Publisher** przy użyciu HTTP za pośrednictwem nowego aktora **Notifier**:
   - **Auction** powiadamia aktora **Notifier** o zmianie stanu aukcji poprzez komunikaty `Notify` zawierające informację o tytue aukcji, aktualnym kupującym i bieżącej cenie.
   - **Notifier** obsługuje komunikaty `Notify` i jest klientem HTTP przekazującym informacje do zewnętrznego serwera **Auction Publisher**.
   - Serwer **Auction Publisher** przyjmuje powiadomienia poprzez komunikaty HTTP POST.
2. (20 pkt) Proszę rozszerzyć aktora **Notifier** o mechanizmy *fault-tolerance*, tak aby działał poprawnie w wypadku awarii zewnętrznego serwera lub awarii sieci. Awarie te nie powinny zakłócać pracy systemu aukcyjnego. Wskazówka: dla każdego wywołania HTTP należy stworzyć nowego aktora **NotifierRequest** nadzorowanego przez aktora**Notifier**.