4. Uvod u Akka.NET

Na ovim vježbama se upoznajemo s Akka.NET tehnologijom. Akka.NET je port popularne Akka tehnologije koja postoji na JVM-u, a nudi API za JAVA-u i Scala-u. Središnji model za istovremeno programiranje u Akka framework-u je istovremenost bazirana na actorima, takozvani actor model. Sama Akka je inspirirana Erlangom. Akka.NET se može koristiti s C#-om i F#-om.

4.1 Zašto actor model?

Ako malo Googlate po internetu, možete pronaći informaciju o tome da se bližimo kraju Mooreovog zakona. Zbog fizičkih ograničenja, ne možemo više ubrzati ni smanjiti procesore te je potreban novi pristup. Taj pristup je počeo prije par godina, a svodi se na dodavanje više jezgri na procesor. S obzirom da su programeri navikli na jednu jezgru, maksimalno dvije, nisu se previše obazirali na stvaranje prorama koji taj višak jezgri mogu iskoristiti. Većinom kada razmišljamo o kodu, razmišljamo, čitamo ga, na način da ga u glavi sekvencijalno izvršavamo, liniju po liniju. Na fakultetima studenti na velikoj većini kolegija radi samo s jednom niti, rade program koji se strogo sekvencijalno izvršava, i to je u redu. Paralelno programiranje - programiranje kojim bi iskoristili ovaj višak jezgri koji nam je danas dostupan je dosta zahtjevnije i sa sobom nosi nove izazove i bug-ove. Najčešći od tih su:

- race condition - situacija u kojoj rezultat ovisni o redosljedu kojim niti pristupaju nekom resursu, što nije poželjno jer rezultat u nikojem slučaju ne možemo predvidjeti, a želimo moći predvidjeti rezultat zbog mogućnosti testiranja koda i validacije da je kod ispravan;
- deadlock -sitaucija u kojoj istovremeni pristup blokira sve niti u ssutavu te nijedna nit ne može nastaviti s obavljanjem svoga posla jer je resurs nedostupan i neće postati dostupan.

S obzirom na sve navedeno, actor model pokušava uvesti razinu abstrakcije, način programiranja, kojim možemo lakše razumjeti što se u kodu događa, validirati da je ispravan, a istovremeno iskoristiti jezgre koje imamo na raspolaganju. S obzirom da je ovo kolegij Raspodijeljeni sustavi, Akka.NET u čijem se središtu nalazi actor model ćemo koristiti kao middleware koji će nam olakšati programiranje komunikacije i određenih drugih aspekata raspodijeljenih sustava, ali za početak se na ovim vježbama moramo upoznati s osnovama.

4.2 Actor model

U računarstvu je actor model jedan od podstojećih modela istovremenog računanja koje tretira "actore" kao osnovnu jedinicu istovremenog izvršavanja. Actor na osnovu poruke koju primi donosi lokalne odluke, stvara nove aktore, šalje poruke i odlučuje treba li poslati odgovor na primljenu poruku.

Organizacija actor sustava podsjeća na hijerarhijsku organizaciju u ljudskom društvu. Naime, točno se zna za što je koji actor zadužen. Komunikacija među actorima podsjeća na ljudsku komunikaciju, jedan actor drugome šalje poruku, poruci treba određena količina vremena da dođe na odredište, a u međuvremenu pošiljatelj može nastaviti raditi svoj posao. Primatelj u trenutku primitka poruke odlučuje kako će reagirati na istu te hoće li poslati odgovor.

Umjesto da ih se zamišlja kao objekte nad kojima se zovu metode, actore je bolje zamišljati kao ljude koji čekaju da prime nekakvu poruku. Actor je formalno definiran kao objekt koji posjeduje:

- identitet;
- ponašanje;
- komunikaciju (asinkronim) slanjem poruka

Prilikom programiranja actora, istome se definiraju poruke koje može primiti te ponašanje za svaki tip poruke. Unutar actora možete koristiti sve što ste do sada radili na ostalim kolegijima.

4.3 Akka.NET

Akka.NET je toolkit za izradu aplikacija vođenih događajima (engl. *event-driven*) koje su istovremene, distribuiranie i otporne na pogreške. Toolkit se može koristiti kao NuGet paket na Mono i .NET platformi. Radi se o portu Akka frameworka sa JVM-a na .NET.

Primjer 4.1: Hello akka main

```
namespace AkkaHelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var actorSystem =
                ActorSystem.Create("AkkaHelloWorld"))
            {
                var greeter =
                    actorSystem.ActorOf<HelloWorldActor>();

                Console.WriteLine("Running on thread: " +
                    Thread.CurrentThread.ManagedThreadId);

                greeter.Tell(new Greet("Marin"));

                actorSystem.WhenTerminated.Wait();
            }
        }
    }
}
```

Primjer 4.2: Hello akka actor

```
using Akka.Actor;
using AkkaHelloWorld.Messages;
using System;
using System.Threading;

namespace AkkaHelloWorld
{
    class HelloWorldActor : ReceiveActor
    {
        public HelloWorldActor()
        {
            Receive<Greet>(s => Console.WriteLine("Hello {0} on
                thread: {1}", s.Who,
                Thread.CurrentThread.ManagedThreadId));
        }
    }
}
```

Primjer 4.3: Hello akka poruka

```
namespace AkkaHelloWorld.Messages
{
    class Greet
    {
        public string Who { get; private set; }

        public Greet(string who)
        {
```

```
9                Who = who;
10           }
11       }
12   }
```

U 4.1-4.3 se nalazi HelloWorld aplikacija za Akka.NET napisan u C#-u. U nastavku je objašnjen primjer.

ActorSystem je hijerarhijska struktura Actora koji imaju istu konfiguraciju. Svi actori žive unutar actor sustava. Actor sustav kreiramo kao u 4.4

Primjer 4.4: Kreiranje actor sustava

```
1            using (var actorSystem =
                 ActorSystem.Create("AkkaHelloWorld"))
2                {
3                    ...
4                }
```

Budući da je actor sustav nešto što u konačnici moramo maknuti iz memorije, instanciramo ga unutar `using`-a. Prvi actor kojeg kreiramo mora biti kreiran direktno iz sustava. U ovome slučaju ga zovemo `greeter`. **Ono što se sprema u varijablu `greeter` nije referenca na samu instancu actora, već njegov IActorRef.** S actorima nikad ne komunicirate direktno! Kada želimo poslati poruku actoru, to činimo uz pomoć metode Tell kojoj pošaljemo instancu poruke. Poruka je najčešće predstavljena nepromjenjivim (engl. *immutable*) objektom. Da bi objekt bio nepromjenjiv, jednom instanciran objekt ne smije mijenjati svoja polja ni svojstva, niti dozvoliti nekome izvana da ga mijenja. Stoga je poruka definirana kao u 4.3. U 4.2 je definirana klasa za actor-a. Ovaj actor nasljeđuje `ReceiveActor`-a koji se nalazi u Akka.Actor imenskom prostoru. Kako bi mogao primati poruke potrebno je uz pomoć metode Receive registrirati kakve poruke actor može primiti. U ovome primjeru smo ih definirali unutar konstruktora klase. Još jedna stvar koju ste mogli primjetiti jest da su actor i main program na različitim nitima. ActorSystem.WhenTerminated() nam daje task na koji možemo čekati. Ono što čekamo jest gašenje actor sustava.

Poruke koje actor prima se spremaju u njegov *mailbox*. **Mailbox** je buffer koji služi za čuvanje poruka. Ovaj buffer je nužan jer actori obrađuju jednu po jednu poruku uvijek. **Poruka** je u akka svijetu jedinica komunikacije koju actori koriste za komunikaciju. Kao što je već rečeno, kada actor pošalje poruku, ne čeka, već nastavlja sa svojim poslom.

4.3.1 IActorRef

Već smo spomenuli da ne komuniciramo direktno s actorima, već preko IActorRef-a. Što je to? IActorRef je *handle* za actor. Kada šaljete poruku actor-u, šaljete je kroz actor sustav, a zatim sustav brine o tome da poruka stigne do odredišta. Ovo je prednost jer:

- Actor sustav zapakira poruku skupa sa svim potrebnim metapodacima te ju actor može prepoznati i raspakirati;
- Omogućuje transparentnost lokacije - to znači da programer ne mora brinuti na kojem stroju se nalazi actor, actor sustav brine o tome.

Najčešće korištena informacija koja se zapakira skupa s porukom jest adresa (IActorRef) pošiljatelja.

4.3.2 Props

Props objekt se može činiti nepotreban, ali je zapravo dosta koristana. Naime, Props je recept za kreiranje actora. Predstavlja konfiguracijsku klasu koja enkapsulira sve potrebne informacije da se

kreira actor određenog tipa. Props se može proširiti da sadrži informacije za puštanje aplikacije u produkciju te ostale postavke koje omogućuju actorima rad na udaljenom računalu.

Props se može stvoriti na nekoliko načina, a dva su navedena u 4.5 i 4.6.

Primjer 4.5: Kreiranje Props-a lambda izrazom

```
1         Props props = Props.Create(() => new MyActor(...),
              "...");
```

Primjer 4.6: Kreiranje Props-a generičkom sintaksom

```
1         Props props = Props.Create<MyActor>();
```

4.4 Primjer čitanja i pisanja

U ovome primjeru ćemo stvoriti dva actora, jedan actor će čitati unose s konzole, a drugi će ispisivati nekakvu poruku.

Kodovi su dani u 4.7-4.10.

Primjer 4.7: Main program

```
1  using Akka.Actor;
2  using AkkaReadWriteConsole.Messages;
3  using System;
4
5  namespace AkkaReadWriteConsole
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             using(var actorSystem = ActorSystem.Create("Sustav"))
12             {
13                 // Kreiramo props te ga posaljemo u ActorOf
                       metodu da kreiramo novog actora
14                 IActorRef writer =
                       actorSystem.ActorOf(Props.Create(() => new
                       WriterActor()));
15                 // Kreiramo reader-a isto kao i writera, samo
                       sto saljemo referencu na writer-a prilikom
                       kreiranja
16                 IActorRef reader =
                       actorSystem.ActorOf(Props.Create(() => new
                       ReaderActor(writer)));
17
18                 // Zapocinjemo izvrsavanje prvog actora
19                 reader.Tell(new ConsoleInput());
20                 actorSystem.WhenTerminated.Wait();
21             }
22         }
23     }
24 }
```

Primjer 4.8: Writer actor

```
using System;
using System.Collections.Generic;
using System.Linq;
using Akka.Actor;
using AkkaReadWriteConsole.Messages;

namespace AkkaReadWriteConsole
{
    class WriterActor : ReceiveActor
    {
        public WriterActor()
        {
            Receive<Vrijednost>(x => IspitajIIsprintaj(x.Broj));
        }

        private void IspitajIIsprintaj(int x)
        {
            if(x % 2 == 0)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("Broj je paran!");
            }
            else
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("Broj je neparan");
            }
        }
    }
}
```

Primjer 4.9: Reader actor

```
using Akka.Actor;
using AkkaReadWriteConsole.Messages;
using System;

namespace AkkaReadWriteConsole
{
    class ReaderActor : ReceiveActor
    {
        private IActorRef _writerActor;

        // Readeru saljemo IActorRef actora koji ce ispisivati
           na ekran.
        public ReaderActor(IActorRef writerActor)
        {
            _writerActor = writerActor;

            // Kada primi poruku tipa ConsoleInput, pozvat ce
               metodu Inputs
            Receive<ConsoleInput>(x => Inputs());
```

```
18          }
19
20          private void Inputs()
21          {
22              try
23              {
24                  Console.WriteLine("Enter a number:> ");
25                  string x = Console.ReadLine();
26
27                  _writerActor.Tell(new Vrijednost(int.Parse(x)));
28              }
29              catch
30              {
31                  Console.WriteLine("Neispravan unos");
32              }
33
34              // Actor moze sebi poslati poruku
35              Self.Tell(new ConsoleInput());
36          }
37      }
38  }
```

Primjer 4.10: Poruke

```
1   namespace AkkaReadWriteConsole.Messages
2   {
3       class ConsoleInput
4       {
5       }
6   }
7
8   namespace AkkaReadWriteConsole.Messages
9   {
10      class Vrijednost
11      {
12          public int Broj { get; private set; }
13
14          public Vrijednost(int broj)
15          {
16              Broj = broj;
17          }
18      }
19  }
```

Svaki actor ima svoju adresu te može sebi poslati poruku. Na ovaj način (primjer 4.9) možemo stvoriti petlju.

Pokretanjem ovog primjera, može se vidjeti da je ispis neuredan, naime, poruka koja nam kaže da nešto unesemo se nalazi prije poruke o broju. Actori sa sobom nose Context. Ovo svojstvo im omogućuje pristup actor sustavu u kojemu su pokrenuti te pristup nekim njegovim svojstvima, između ostalog i svojstvu Scheduler koje posjeduje metodu ScheduleTellOnce koja će nakon određenog vremenskog odmaka poslati poruku.

Primjer 4.11: ScheduleTellOnce

```
Context.System.Scheduler.ScheduleTellOnce(TimeSpan.FromSeconds(2),
    Self, new ConsoleInput(), Self);
```

Ova linija će zatražiti od sustava da (objašnjeni su argumenti):

1. pošalje poruku jednom za dvije sekunde
2. pošalje tu poruku sebi
3. poruka koju će poslati
4. tko će biti naveden kao pošiljatelj.

Kako gasimo actor sustav? - Context.System.ShutDown() - ako želimo sustav izgasiti iz actora.

4.5 Što je s porukama za koje nismo naveli ponašanje?

Nitko ne brani nekome klijentu da našem actoru pošalje poruku koju nismo predvidjeli. Što napraviti u toj situaciji?

Budući da naš actor nasljeđuje klasu ReceiveActor, ima metodu koja se zove Unhandled. Ako overrideamo tu metodu, možemo kontrolirati način na koji će naš actor postupati prema porukama koje nisu definirane. Vodite računa da u slučaju overrideanja ove metode i dalje ostavite poziv na baznu metodu (`base.Unhandled()`).

4.6 Zadatci

1. Uz pomoć Akka.NET-a napravite 2 actora koji će se međusobno dopisivati porukama Ping i Pong. Jedan će primati poruku Ping, a odgovarati s Pong, dok će drugi obrnuto.
2. Izmijenite primjer čitanja i pisanja na način da ReaderActor može primiti porukue "Shut-Down" nakon čega započme gašenje cjelokupnog actor sustava.
3. Isti primjer nadogradite na način da dodate novog actora kojeg ćete nazvati ValidationActor. Validacijski actor prima kao argument funkciju za validaciju (po vašoj volji, u primjeru je to provjera je li broj paran) i actora kojemu šalje poruku (writer). U slučaju da je validacija prošla, actor mora poslati writer-u poruku s brojem i reader-u (onome koji mu je poslao poruku) poruku o uspješnoj validaciji. U suprotnom validator ne šalje poruku writeru, ali odgovara readeru da validacija nije prošla. U oba slučaja reader ispisuje odgovarajuću poruku na ekran te nastavlja čekati unos od korisnika.
4. Napravite actor program u kojemu će jedan glavni actor izbrojiti koliko datoteka se nalazi u zadanom folderu. Za svaku datoteku actor stvara dijete (Context.ActorOf()) koje putem poruke dobije zadatak da pročita sadržaj datoteke, izbroji broj pojedinih samoglasnika te rezultat vrati roditelju. Roditelj nakon što primi rezultate sve svoje dijece ispiše konačan broj svakog od samoglasnika u datotekama. Dijeca po završetku posla i nakon slanja informacija roditelju zaustavljaju sebe pozivom Context.Stop metode. Metoda kao argument prima IActorRef actora kojeg je potrebno zaustaviti.