SPRAWOZDANIE

Zaawansowane aplikacje WWW

Zadania okresowe

19.01.2016

Karol Suwalski 125NCI B

https://github.com/SuwalskiKarol/orw.git

Czym jest QUARTZ.NET?

Czasami potrzeba nam prostego narzędzia, które będzie wykonywało jakieś zadania w określonych ramach czasowych. Można użyć prostego Timeraz .NET Framework, ale ma on dość ograniczone możliwości. Na przykład, stan zadań nie może być zapisany w bazie danych. Quartz.NET jest lekką biblioteką, która nada się do prostych przypadków, dla których jednak czysty, standardowy timer ma zbyt małe możliwości.

NajważniejszeklasyQuartza:
-IScheduler - Główne API do interakcji z harmonogramem.
- IJob – Implementuje elementy, które mają być wykonane przez harmonogram.
- IJobDetail – używany do określania instancji Jobsów.
- ITrigger - elementem, który określa harmonogram, na którym dana praca zostanie wykonana.
- JobBuilder - Używany do definiowania lub budowania JobDetail, który definiuje instancje Jobsów
- TriggerBuilder - Używany do budowania Triggerów.
- Listeners – są to obiekty utworzone do wykonania czynności na podstawie zdarzeń mających miejsce w ramach harmonogramu.

Jak wygląda budowa Quartz?
Prosty scheduuler wypisujący nam, co 10 sec aktualną godzinę.

Scheduler
Sercem każdego harmonogramu jest obiekt klasy. Zawiera on informacje o zadaniach oraz czasie ich wykonania. Uzyskujemy go za pomocą fabryki. Fabrykę zaimplementujemy samodzielnie wykorzystując kompozycję i klasę StdSchedulerFactory.
Obiekt scheduler po utworzeniu wymaga jeszcze wywołania metody start().

```
publicclassMySchedulerFactory implements SchedulerFactory
{
privateSchedulerFactoryschedulerFactory = newStdSchedulerFactory();

@SuppressWarnings("unchecked")
public Collection getAllSchedulers() throws SchedulerException
{
returnschedulerFactory.getAllSchedulers();
}

public Scheduler getScheduler() throws SchedulerException
{
    Scheduler scheduler = schedulerFactory.getScheduler();
if (!scheduler.isStarted()) {
scheduler.start();
    }
return scheduler;
}
public Scheduler getScheduler(StringschedName) throws SchedulerException
{
    Scheduler scheduler = schedulerFactory.getScheduler(schedName);
if (!scheduler.isStarted()) {
scheduler.start()
returnscheduler;
```

Zadania

Zadania, reprezentują konkretne czynności do wykonania. Tworzenie zadań polega na implementacji interfejsu Job.

```
publicclassWypisanieCzasuJob implements Job
{


publicvoid execute(JobExecutionContext context)
throwsJobExecutionException
{
    Calendar calendar = GregorianCalendar.getInstance();

System.out.println("Mamy godzinę: "
+ calendar.get(Calendar.HOUR_OF_DAY) + ":"
                                + calendar.get(Calendar.MINUTE) + ":"
                                + calendar.get(Calendar.SECOND));
}
}
```

Jak widać metoda execute(JobExecutionContext) to jedyna metoda, którą musimy zaimplementować. JobExecutionContext zawiera informacje takie jak nazwa zadania, data kolejnego wykonania, licznik wykonań. Za jego pomocą można też przekazywać parametry do zadania.

Wyzwalacze

Czas wykonania zadania określamy za pomocą obiektów Trigger. Biblioteka dostarcza kilka standardowych implementacji dla wyzwalaczy są to miedzy innymi CronTrigger i SimpleTrigger. Oczywiście można samodzielnie zaimplementować własny wyzwalacz, ale zazwyczaj nie ma takiej potrzeby. W przypadku gdy chcemy uzyskać jakiś standardowy interwał na przykład godzinny lub dniowy można skorzystać z klasy TriggerUtils, która zawiera odpowiednie metody fabrykujące.

Uruchomienie

Skoro mamy już przygotowane zadanie i umiemy uzyskać scheduler to połączmy wszystkie te elementy w jednym programie.

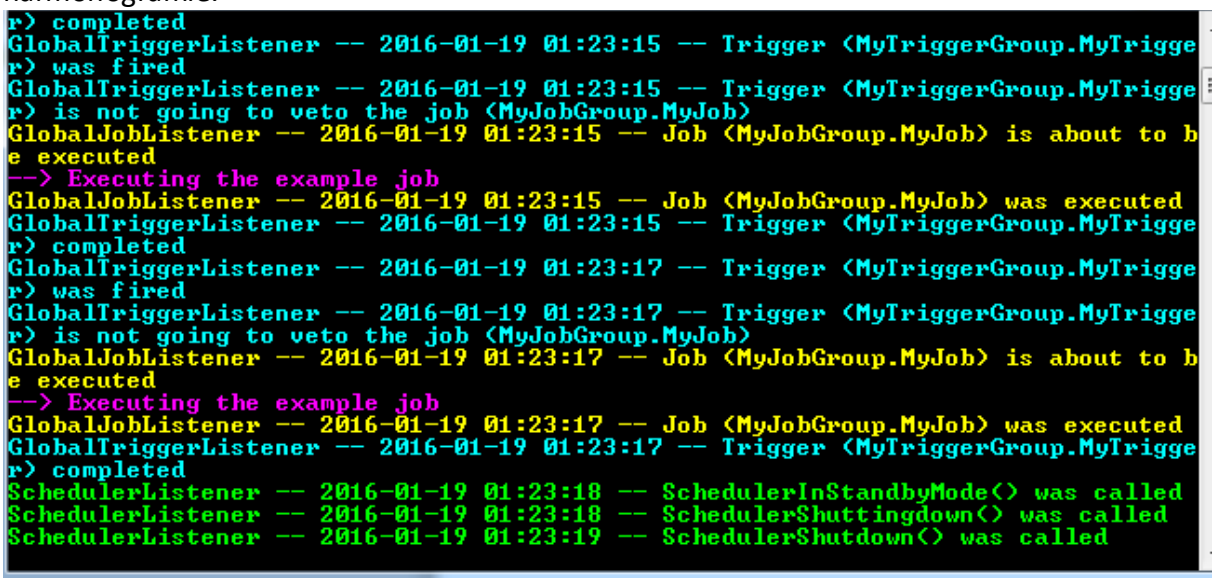
```
publicclassHarmonogram
{
publicstaticvoid main(String[] args)
    {
SchedulerFactoryschedulerFactory = newMySchedulerFactory();
try
        {
            Scheduler scheduler = schedulerFactory.getScheduler();
JobDetailjobDetail = newJobDetail("przykład", null,
WypisanieCzasuJob.class);
                        Trigger trigger = newSimpleTrigger("simple", null,
SimpleTrigger.REPEAT_INDEFINITELY, 10000l);
scheduler.scheduleJob(jobDetail, trigger);
                } catch (SchedulerException e) {
e.printStackTrace();
}
        }
```

JobDetail zawiera konfigurację zadania. Pierwszy parametr to nazwa zadania, drugi nazwa grupy (w przypadku podania wartości null będzie to grupa domyślna), a trzeci to klasa implementująca Job. Tworzenie SimpleTrigger wymaga, podobnie jak w przypadku JobDetail, podania nazwy wyzwalacza, nazwy grupy (w przypadku podania wartości null będzie to grupa domyślna), dodatkowo podajemy ilość wywołań oraz czas pomiędzy poszczególnymi wywołaniami.

Nasłuchiwanie

Quartz udostępnia nam dwa bardzo przydatne interfejsy służące do obserwowania wykonania zadań. Pierwszy z nich to JobListener, który zawiera metody uruchamiane przez scheduler przed i po wykonaniu zadania oraz w momencie, gdy zadanie zostało anulowane. Drugi to TriggerListener, który zawiera podobne metody, ale dla wyzwalacza.

Przykład wykorzystujący listenery:
Dzięki listenerom mamy możliwość zaobserwować, co się aktualnie dzieje w naszym harmonogramie.



Scheduler listener:

```
classSchedulerListener : ISchedulerListener
    {
privatestaticvoid Write(string text, paramsobject[] args)
        {
Console.ForegroundColor = ConsoleColor.Green;
Console.WriteLine(text, args);
Console.ResetColor();
        }

publicstring Name { get { return"SchedulerListener"; } }

publicvoidJobScheduled(ITrigger trigger)
        {
Write("{0} -- {1} -- JobScheduled() was called", Name, DateTime.Now);
        }

publicvoidJobUnscheduled(TriggerKeytriggerKey)
        {
Write("{0} -- {1} -- JobUnscheduled() was called", Name, DateTime.Now);
        }

publicvoidTriggerFinalized(ITrigger trigger)
        {
```

```
Write("{0} -- {1} -- TriggerFinalized() was called", Name, DateTime.Now);
}
```

Job listener:

```
classJobListener : IJobListener
    {
privatestaticvoid Write(string text, paramsobject[] args)
        {
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine(text, args);
Console.ResetColor();
        }

publicstring Name { get { return"GlobalJobListener"; } }

publicvoidJobToBeExecuted(IJobExecutionContext context)
        {
Write("{0} -- {1} -- Job ({2}) is about to be executed", Name, DateTime.Now,
context.JobDetail.Key);
        }

publicvoidJobWasExecuted(IJobExecutionContext context,
JobExecutionExceptionjobException)
        {
Write("{0} -- {1} -- Job ({2}) was executed", Name, DateTime.Now,
context.JobDetail.Key);
}
```

Triggerlistener:

```
classTriggerListener : ITriggerListener
    {
privatestaticvoid Write(string text, paramsobject[] args)
        {
Console.ForegroundColor = ConsoleColor.Cyan;
Console.WriteLine(text, args);
Console.ResetColor();
        }

publicstring Name
        {
get { return"GlobalTriggerListener"; }
        }

publicvoidTriggerFired(ITrigger trigger, IJobExecutionContext context)
        {
Write("{0} -- {1} -- Trigger ({2}) was fired", Name, DateTime.Now, trigger.Key);
}
```

Jedną z metod konfigurowania naszych obiektów scheduler, triggers czy jobs jest zapisywanie opcji konfiguracyjnych w pliku XML:

```
<schedule>
<job>
<name>MyJob</name>
<group>MyJobGroup</group>
<description>My example job</description>
<job-type>QuartzWithXmlConfiguration.ExampleJob, QuartzWithXmlConfiguration</job-type>
</job>
<trigger>
<cron>
<name>MyTrigger</name>
<group>MyTriggerGroup</group>
<job-name>MyJob</job-name>
<job-group>MyJobGroup</job-group>
<misfire-instruction>DoNothing</misfire-instruction>
<cron-expression>0/1 * * * * ?</cron-expression>
</cron>
</trigger>
</schedule>
```

Nincject

Największy problem Quartzajest zewstrzykiwaniem zależności w konstruktoryjobów. *dependencyinjection* (DI) . Z pomocą przychodzi paczka Ninject

Budowa modułu QuartzNinjectModule:

```
publicclassQuartzNinjectModule : NinjectModule
{
publicoverridevoid Load()
    {
        Bind<ISchedulerFactory>().To();
        Bind<IScheduler>().ToMethod(c = &gt;
c.Kernel.Get().GetScheduler()).InSingletonScope();
}
}
```

W module korzystamy z fabryki planera, którą musimy sobie sami stworzyć.Ostatnią klasą, jaką musimy dodać do projektu to NinjectJobFactory:

```
publicclassNinjectJobFactory : IJobFactory
{
privatestaticreadonlyILog Log = LogManager.GetLogger(typeof(NinjectJobFactory));

privatereadonlyIKernel _kernel;

staticNinjectJobFactory()
    {
    }

publicNinjectJobFactory(IKernel kernel)
    {
        _kernel = kernel;
    }

publicIJobNewJob(TriggerFiredBundle bundle, IScheduler scheduler)
    {
IJobDetailjobDetail = bundle.JobDetail;
TypejobType = jobDetail.JobType;
try
        {
Log.Debug($"Producing instance of Job '{jobDetail.Key}', class={jobType.FullName}");
```

```
return _kernel.Get(jobType) asIJob;
        }
catch (Exception ex)
        {
thrownewSchedulerException($"Problem instantiating class '{jobType.FullName}'", ex);
        }
    }

publicvoidReturnJob(IJob job)
{
    }
}
```

Po dodaniu tych klas do projektu, nasza klasa Job mająca w konstruktorze zależność będzie działać o ile poprawnie zainicjujemy Scheduler.