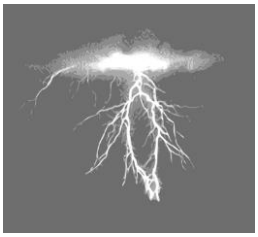

Introduction à Apache STORM

Premier programme…

Version 1.0

Rédacteurs V1 : Philippe Lacomme (placomme@isima.fr), Raksmey Phan (phan@isima.fr)

Date : 22 août 2015

Rédacteurs V2 : Philippe Lacomme (placomme@isima.fr), Raksmey Phan (phan@isima.fr), étudiants en projets ISIMA (Soriano Baptiste et Zouggari Yannis)

Date : 1 juin 2016

Installation réalisée sur : Ubuntu 15.04

Environnement : Vmware

Licence :

Objectifs :

3 objectifs dans ce tutoriel :

- apprendre à configurer et installer l'environnement de développement Java pour Storm ;
- réaliser un premier programme Java pour compter l'occurrences de mots dans un texte ;
- appréhender le principe d'exécution d'une architecture Storm.

Etape 1. Installer Netbeans

Il faut se procurer une version récente de Netbeans en consultant le site :

https://netbeans.org/

A l'heure où ce tutoriel est écrit, la dernière version est la 8.0.2 (figure 1).

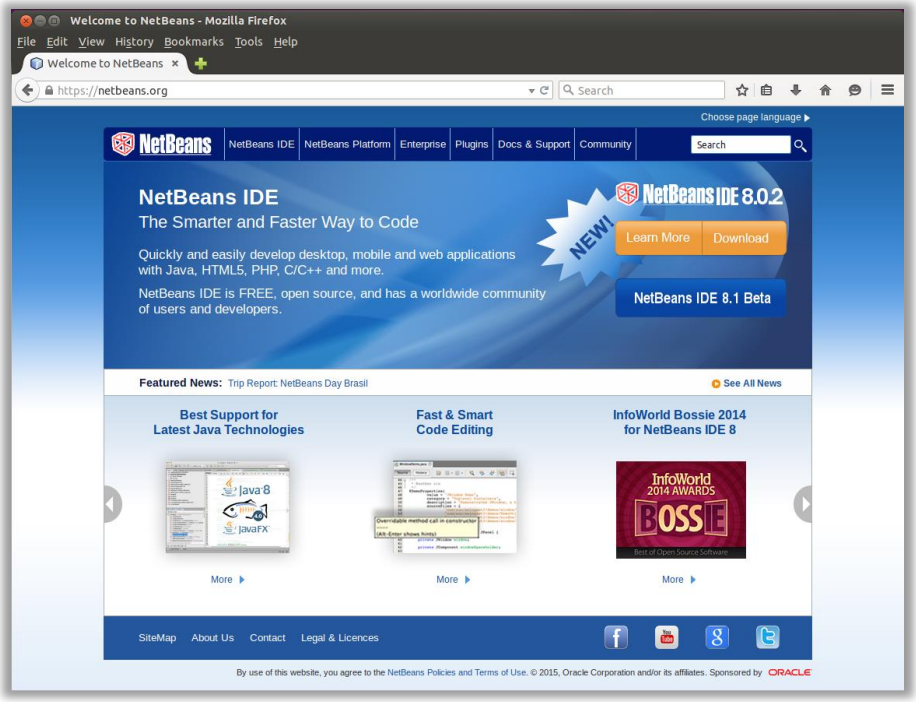

Figure 1. Accès au site de Netbeans

Bien que cette version offre plus de possibilités que celles nécessaires pour réaliser le tutoriel, il est recommander de sélectionner la version **All** qui est la plus complète (figure 2).

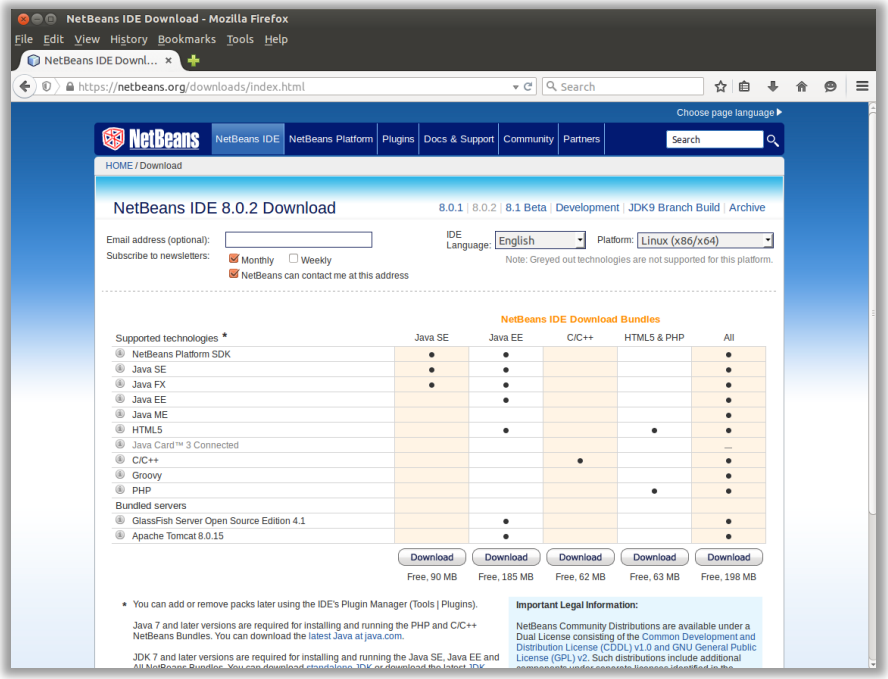

Figure 2. Choix de la version

Il faut sauvegarder en local la version qui, sous Linux, se présente sous la forme d'un fichier **.sh** (figure 3).

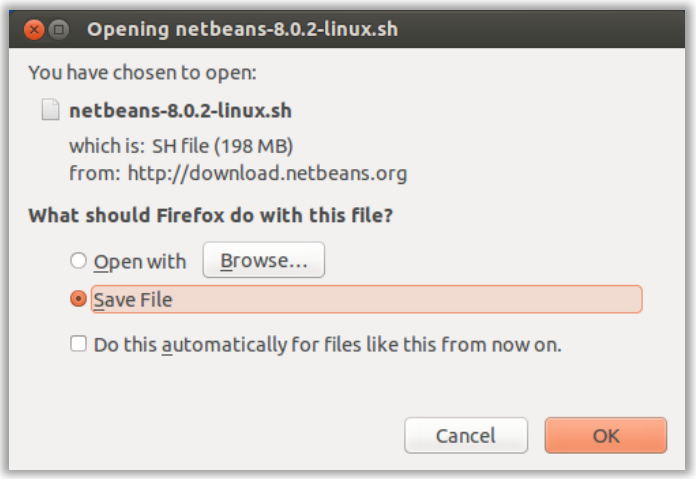

Figure 3. Le fichier d'installation pour Linux

Le plus simple consiste à recopier ce fichier dans le répertoire nommé **Installation** qui normalement se trouve sur le bureau (figure 3).

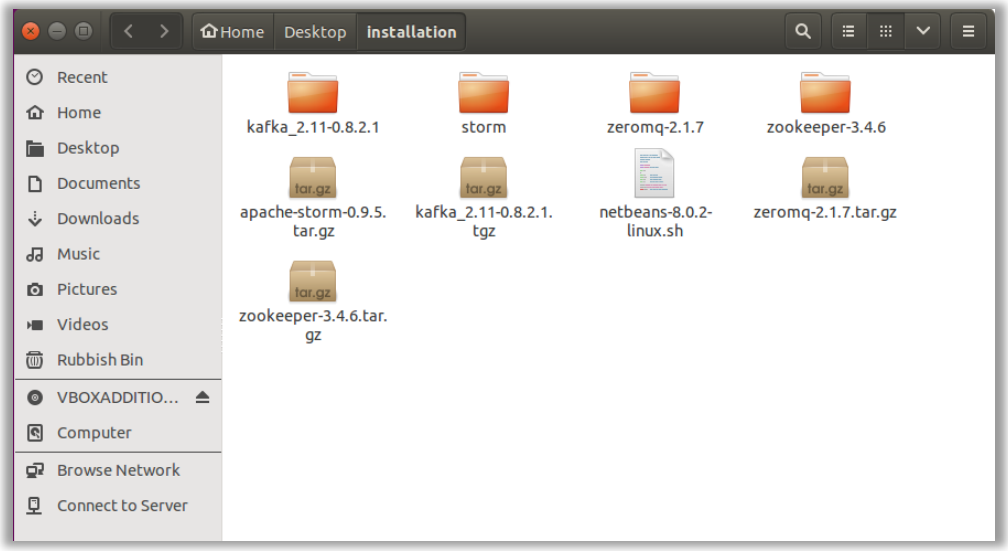

Figure 4. Le fichier d'installation pour Linux dans le répertoire **installation**

Pour lancer l'installation il faut :

- modifier les droits du fichier en rendant ce fichier exécutable : on peut pour cela les passer en 777 avec la commande : **chmod 777 xxxxxxxxxxxxxxxx.sh**

- lancer l'installation avec la commande : **./xxxxx.sh**

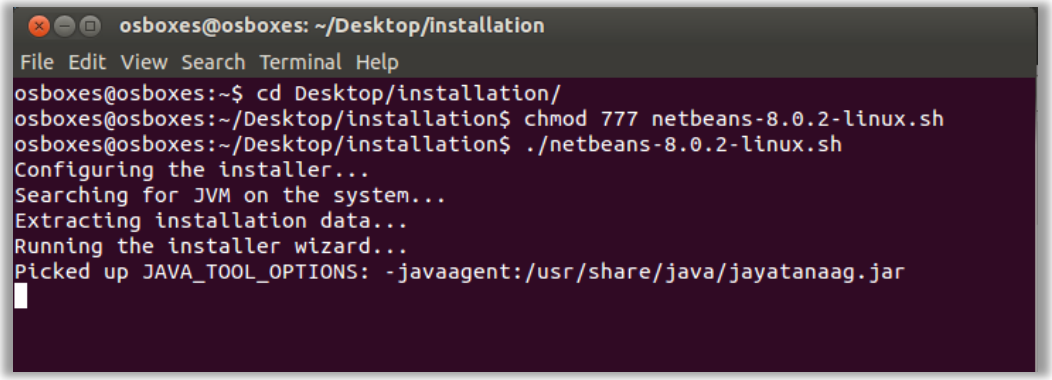

Figure 5. Démarrage de l'installation

Le plus difficile étant terminé, il faut se laisser guider au travers des 3/4 écrans qui nécessitent des validations par le bouton suivant (figure 6).

Cependant, veuillez à bien indiquer le bon dossier du jdk de java.

Figure 6. Installation… en cours

Etape 2. Principe d'un programme utilisant Storm

Une **topologie** est la manière dont les éléments **Spout** et **Bolt** communiquent entre eux pour fournir le résultat final Ces éléments communiquent va des **Stream** (flux). On définit les éléments suivants :

- un **Spout** est l'équivalent d'un adapteur qu'on retrouve dans la plus part des framework de développement (Visual Studio par exemple) qui se connecte à une source de données et qui va extraire de cette source des **tuples** à destination de un ou plusieurs **Bolt**.

- un **Bolt** est une fonction de calcul qui est appliquée à un tuple et qui va produire en sortie un autre flux. Classiquement, un **Bolt** réalise des opérations de filtrage, des agrégations ou des calculs.

- un **Stream** est un flux composé de tuple de type **<clé; valeur>.**

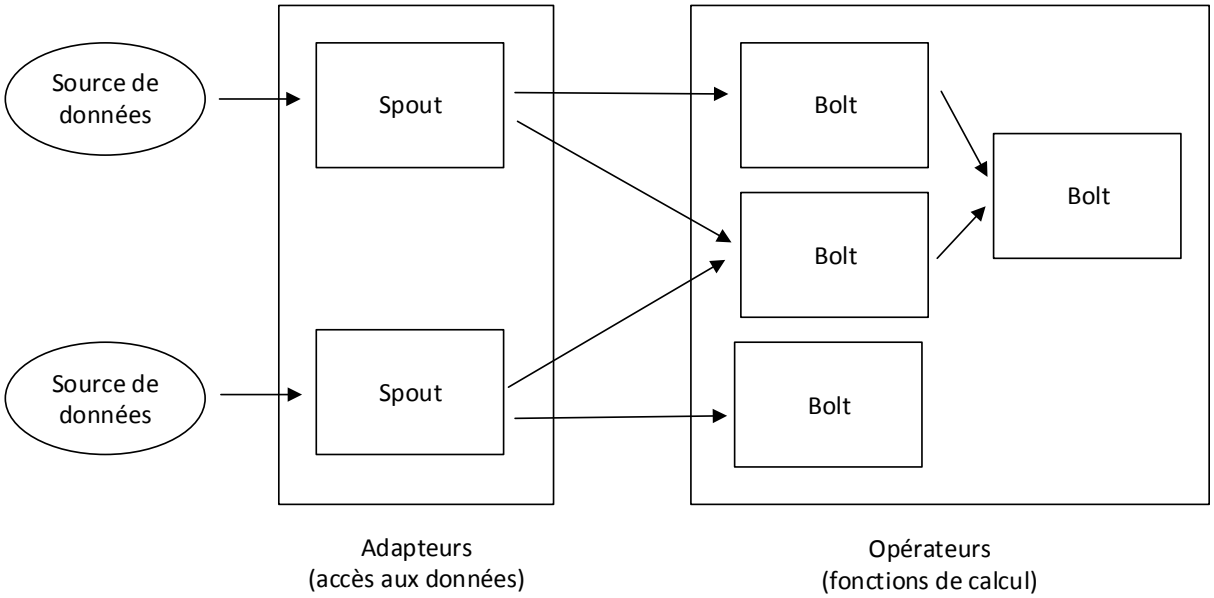

Figure 7. Notion de **Topologie**

Etape 3. Installer un exemple Java déjà existant

Il existe quelques exemples Java disponibles avec un code qu'on peut facilement recompiler et exécuter. Un de ces codes se trouve sur github à l'adresse suivante :

https://github.com/apache/storm/tree/master/examples/storm-starter

On peut consulter la page web du code qui offre la possibilité de parcourir les fichiers du projet séparément (figure 8).

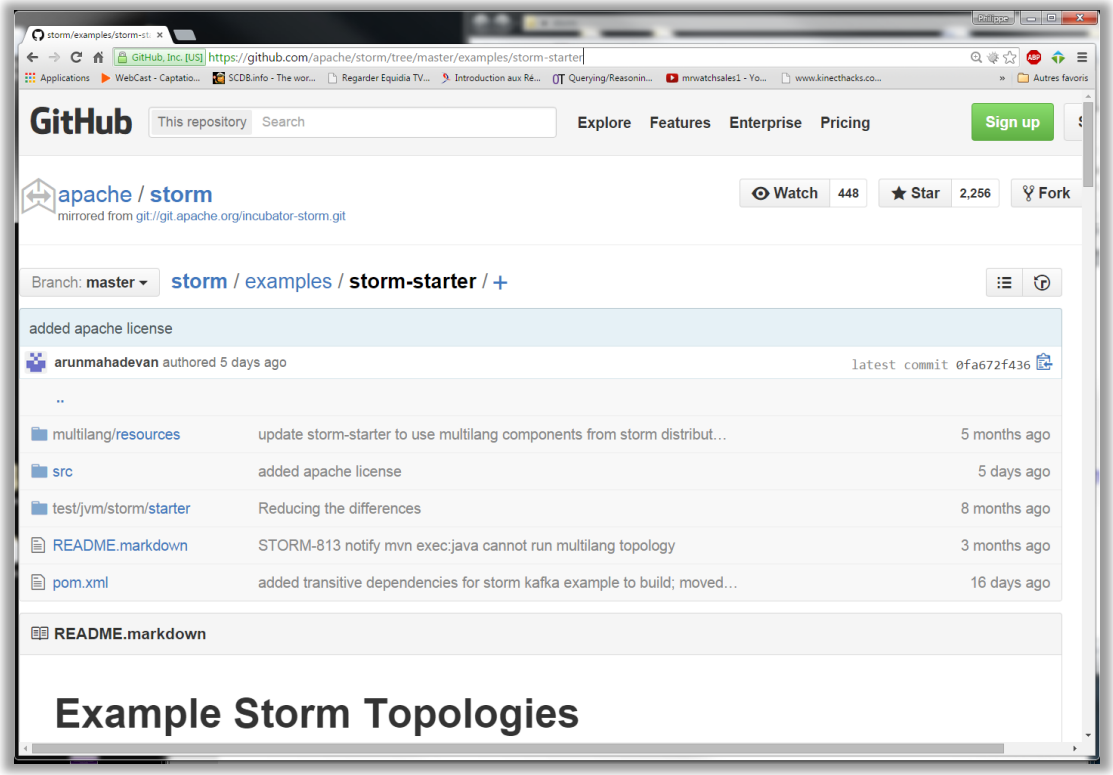

Figure 7. La page github du code Java

Si vous souhaitez tester ce code, créez un répertoire dans le dossier **installation** (figure 8) et réaliser une copie du dépôt avec la commande git (figure 9).

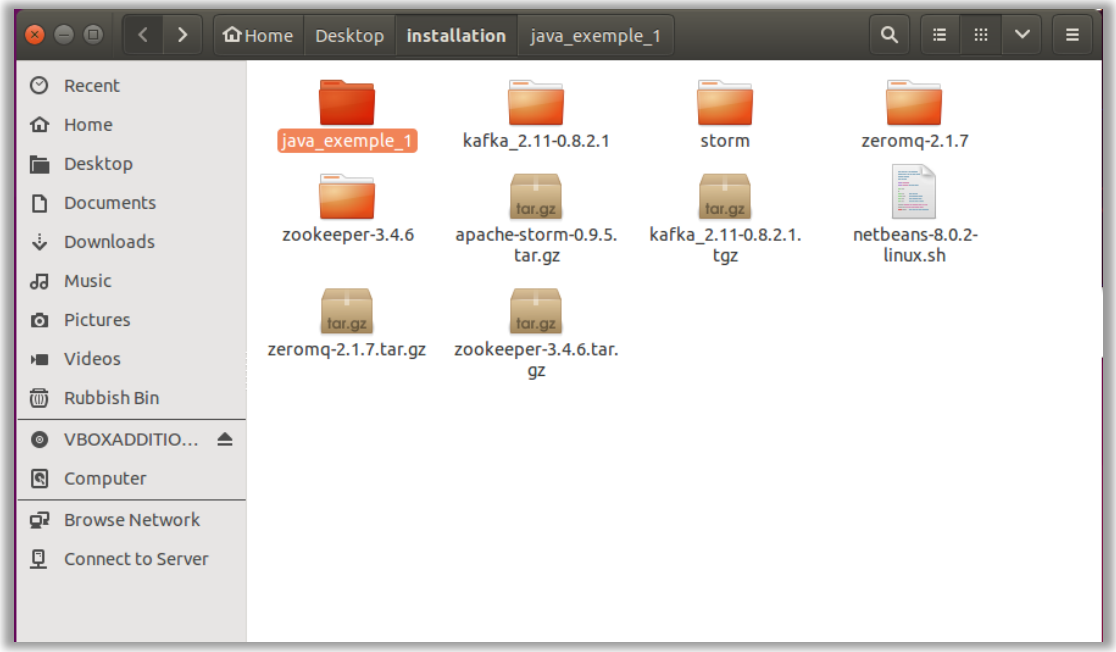

Figure 8. Un répertoire pour l'exemple Java

```
osboxes@osboxes:~/Desktop/installation$ cd java_exemple_1/
osboxes@osboxes:~/Desktop/installation/java_exemple_1$ git clone git://github.com/apache/storm.git
Cloning into 'storm'...
remote: Counting objects: 49670, done.
remote: Total 49670 (delta 0), reused 0 (delta 0), pack-reused 49670
Receiving objects: 100% (49670/49670), 21.09 MiB | 2.54 MiB/s, done.
Resolving deltas: 100% (23988/23988), done.
Checking connectivity... done.
osboxes@osboxes:~/Desktop/installation/java_exemple_1$
```

Figure 8. Recopie du dépôt

Avant d'utiliser ce code Java, il faut se rendre dans le répertoire Storm (figure 9) et taper la commande suivante : **mvn clean install -DskipTests=true**.

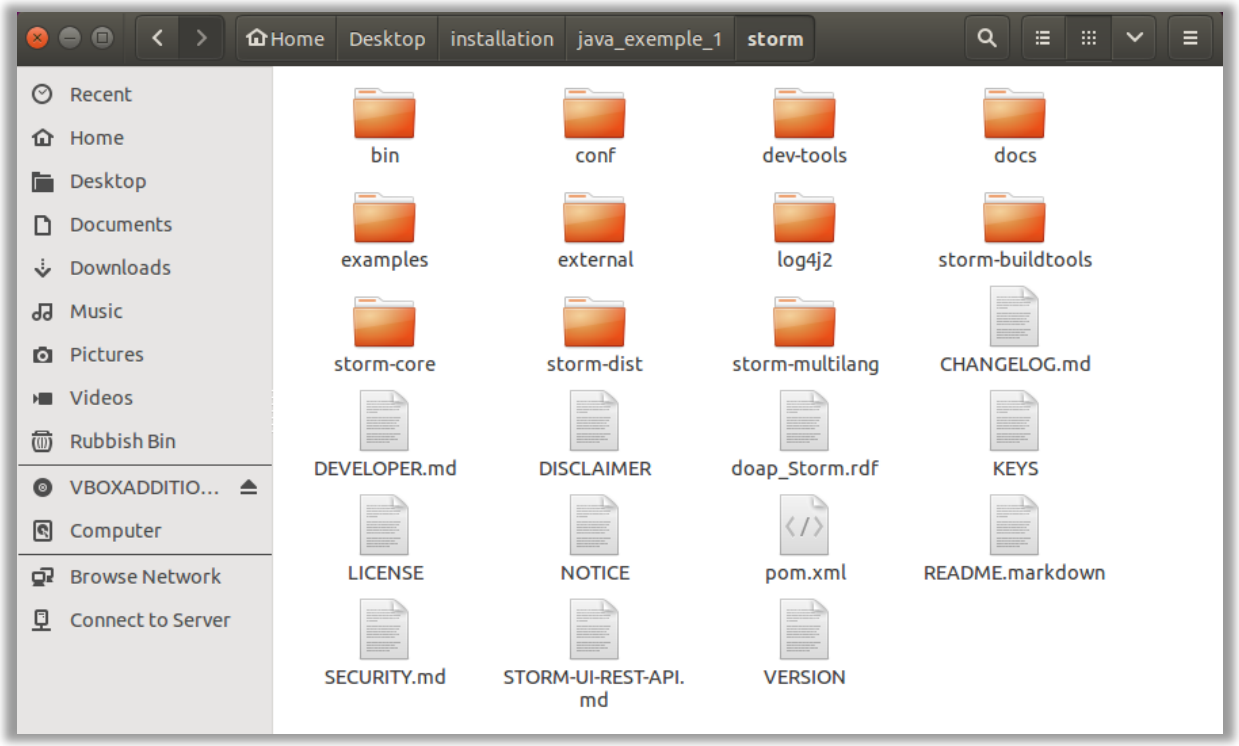

Figure 9. Contenu du répertoire Storm

Etape 4. Réalisation d'un code Java complet : compter l'occurrence des mots dans un texte

1. Définition d'une topologie

Pour compter le nombre de mots dans un fichier, la topologie nécessaire se compose (figure 10) :

- d'un **Spout** qui va découper le fichier en phrases ;

- de trois **Bolts** : l'un pour découper des phrases en mot, l'un pour compter les mots et l'autre pour réaliser l'affichage du résultat.

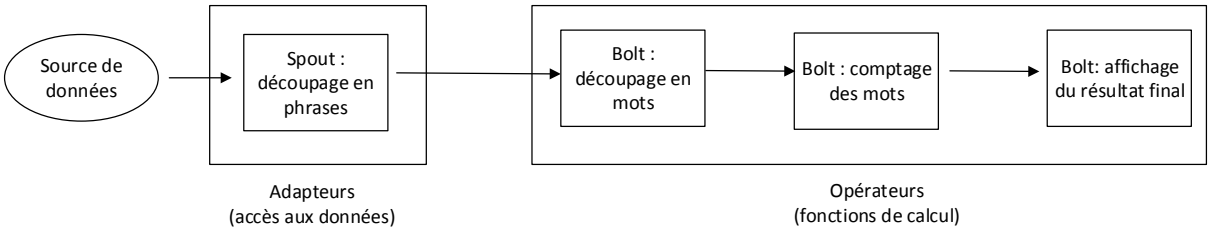

Figure 10. Topologie utilisée

Pour des raisons de simplicité, la manipulation du fichier a été supprimée et les chaînes de caractères permettant d'illustrer le propos sont stocker "en dur" dans le Spout c'est-à-dire dans l'adapteur. Cela donne le schéma de la figure 11.

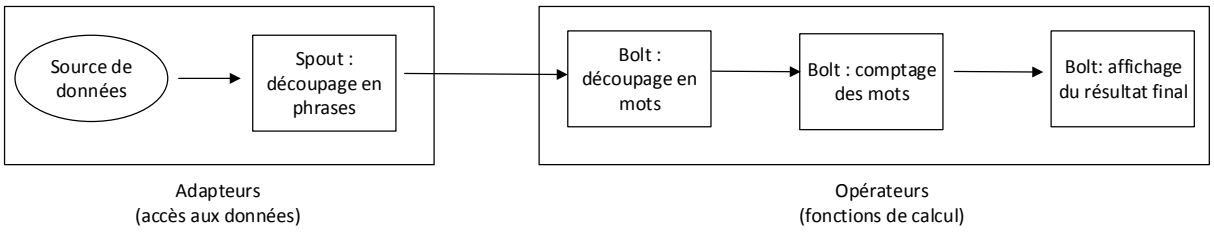

Figure 11. Topologie utilisée (sans fichier de données)

Les flux qui sont utilisés dans cette topologie sont de 3 types (figure 12) :

- type 1. les "**flux de ligne**" de la forme {"sentences":"xxxxxxxxxx"} sont sont crées par le Spout et récupérés par le Bolt qui découpe les mots. Ceci correspond à des lignes qui seraient lues dans un fichier par exemple.

- type 2. les "**flux de mots**" de la forme {"word":"xxxxxxxxxx"} sont crées par le Bolt de découpage et récupérés par le Bolt de comptage. Ce flux représente les mots d'une phrase.

- type 3. les "**flux de compteurs**" de la forme {"word":"**xxxxxxxxxx**","count": **yy**} sont créés par le Bolt de comptage et associe à chaque mot un tuple de la forme <"count", yy> où yy est un nombre entier représentant le nombre de fois que le mot "**xxxxxxxxxx**" apparaît.

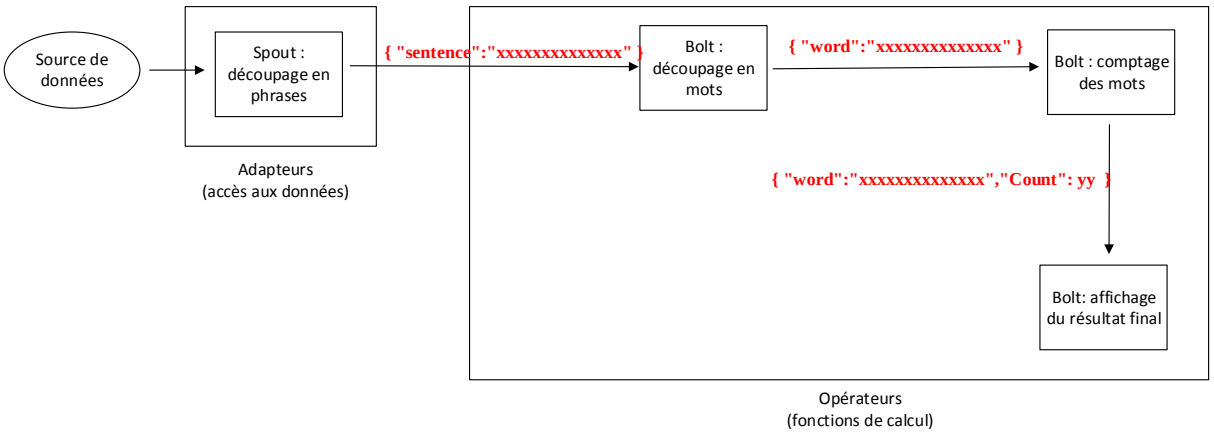

Figure 12. Les flux utilisés dans la topologie

2. Création d'un projet Java

Il faut créer un projet de type **JavaApplication** (figure 13) avec comme nom par exemple **DemoStorm** (figure 14).

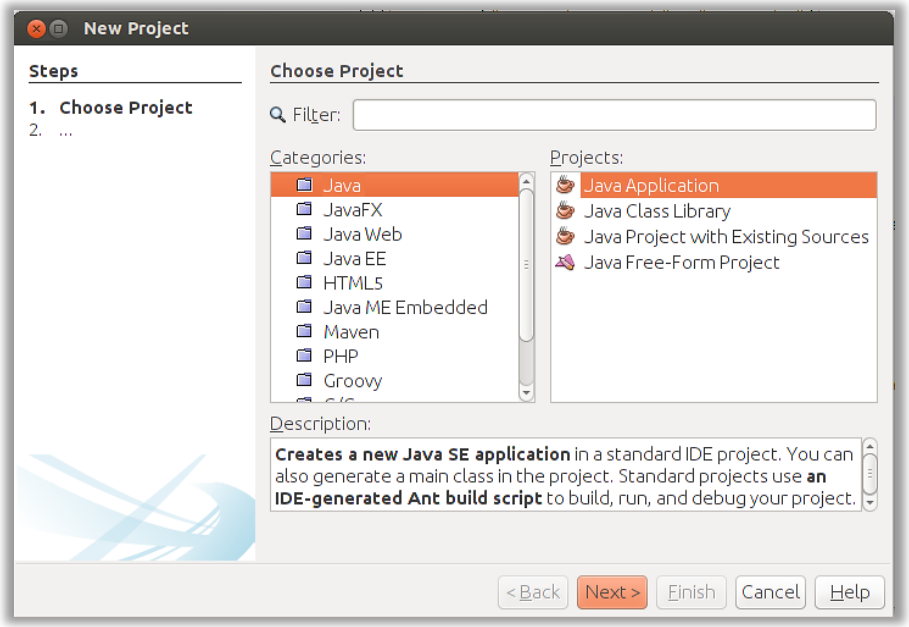

Figure 13. Création d'un projet Java

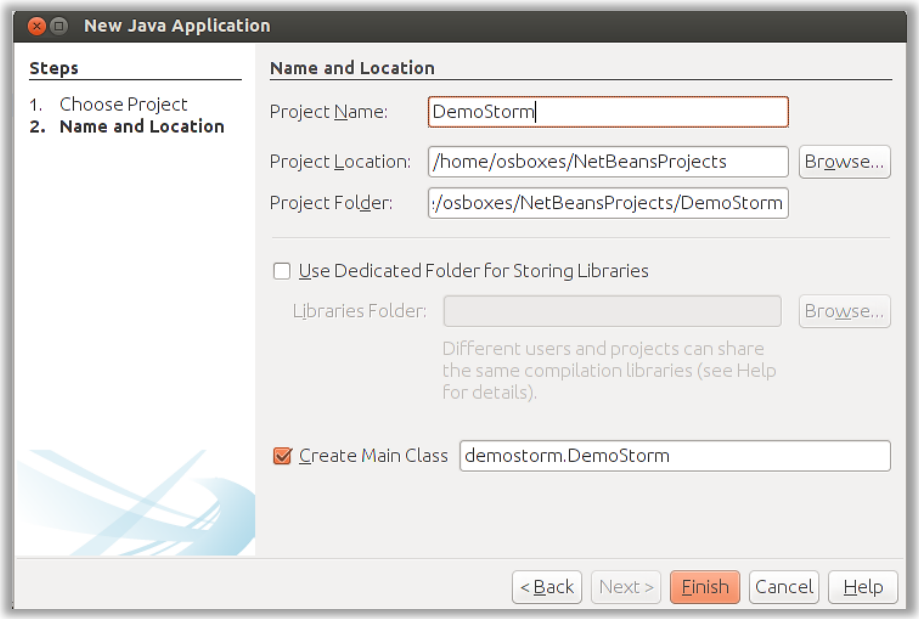

Figure 14. Définition du projet Java

Une fois la création du projet réalisée, il faut ajouter les librairies **Storm** au projet. Il faut faire un clic droit sur **Libraries** et sélectionner **Add JAR/Folder** (figure 15).

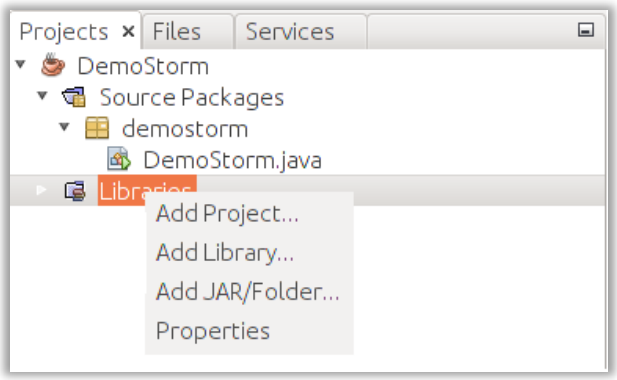

Figure 15. Ajout des librairies Storm

Les librairies se trouvent dans le répertoire **/home/Desktop/Installation/Storm/lib** et il faut toutes les sélectionner (figure 16).

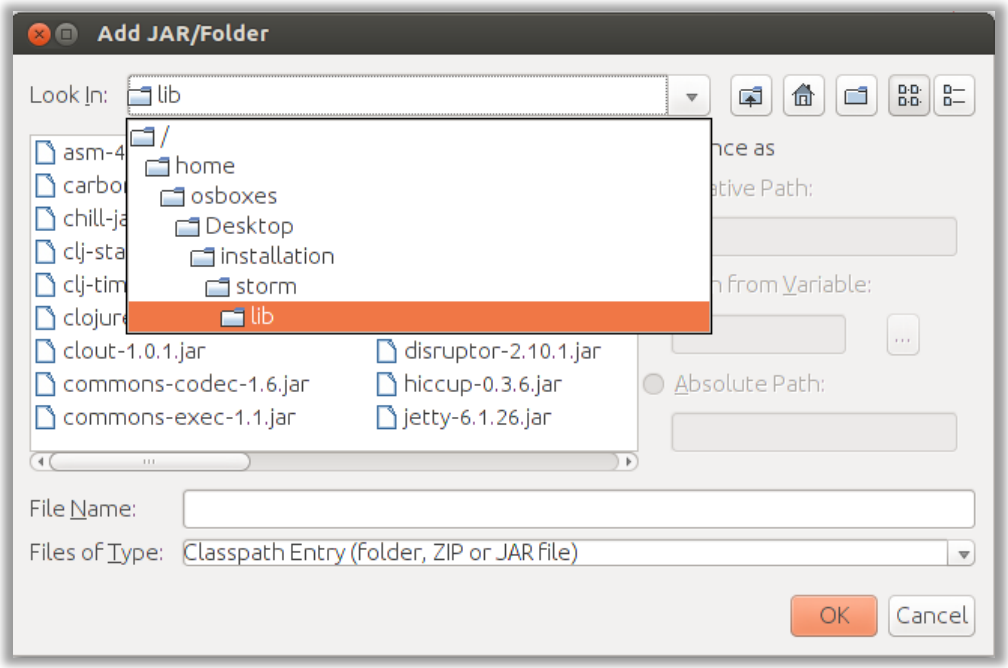

Figure 16. Ajout des librairies Storm

Les fichiers Jar ajoutés doivent apparaître dans la section **Libraries** du projet (figure 17).

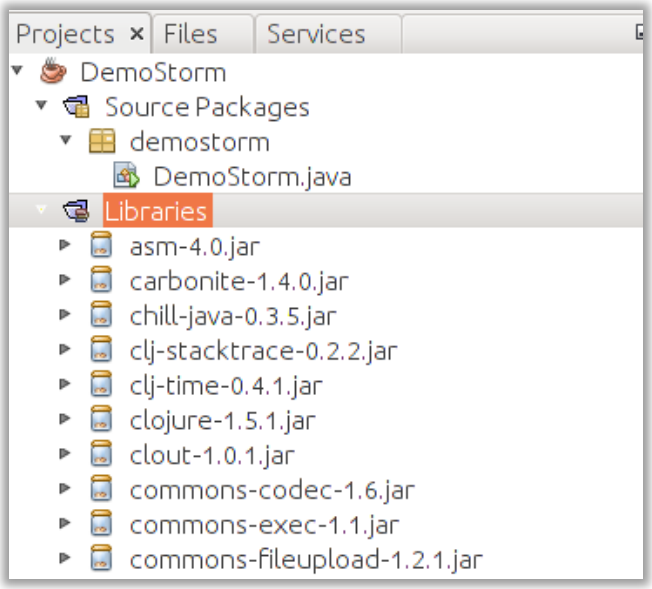

Figure 17. Les librairies Storm dans le projet

Remarque :

Nous supposerons que le lecteur dispose d'une classe permettant de faciliter la manipulation des Threads. Cette classe à minima implémentera deux procédures permettant de créer des petites attentes dans les threads. On peut pour cela s'inspirer de la classe **Utils** donnée ci-dessous. Si cette classe n'est pas disponible sur la plateforme de développement, il faut l'ajouter dans le projet.

```
public class Utils {

    public static void waitForSeconds(int seconds) {
        try {
            Thread.sleep(seconds * 1000);
        } catch (InterruptedException e) {

        }
    }

    public static void waitForMillis(long milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {

        }
    }
}
```

3. Création du Spout

Le Spout a pour objectif de créer un flux (stream) de couples <clé;valeur> de la forme

{ "sentence":"**xxxxxxxxxxxxxx**" }

où "**xxxxxxxxxxxxxx**" est une phrase. On obtient par exemple un stream comme celui-ci :

{ "sentence":"Mon cher enfant que j'ai vu dans ma vie errante," }

Ceci correspond à la première fonctionnalité représentée sur le schéma de la figure 18.

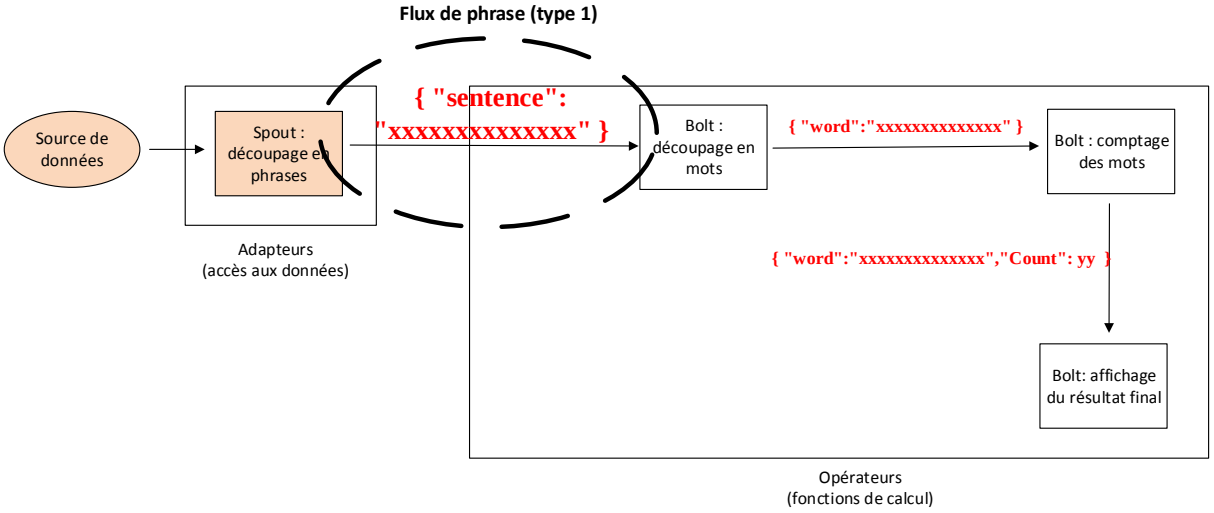

Figure 18. Création du Spout

Pour tester le programme, il faut disposer d'un ensemble de lignes suffisamment long. Ici, pour donner une touche poétique à ce tutoriel, il a été choisi le poème "**A un passant**" de Paul Verlaine :

```
"A un passant",
"Mon cher enfant que j'ai vu dans ma vie errante,",
"Mon cher enfant, que, mon Dieu, tu me recueillis,",
"Moi-même pauvre ainsi que toi, purs comme lys,",
"Mon cher enfant que j'ai vu dans ma vie errante !",
"Et beau comme notre âme pure et transparente,",
"Mon cher enfant, grande vertu de moi, la rente,",
"De mon effort de charité, nous, fleurs de lys !",
"On te dit mort... Mort ou vivant, sois ma mémoire !",
"Et qu'on ne hurle donc plus que c'est de la gloire",
"Que je m'occupe, fou qu'il fallut et qu'il faut...",
"Petit ! mort ou vivant, qui fis vibrer mes fibres,",
"Quoi qu'en aient dit et dit tels imbéciles noirs",
"Compagnon qui ressuscitas les saints espoirs,",
"Va donc, vivant ou mort, dans les espaces libres !",
"Paul Verlaine"
```

Il faut ajouter une classe nommée **SentenceSpout** dans le projet (figure 19).

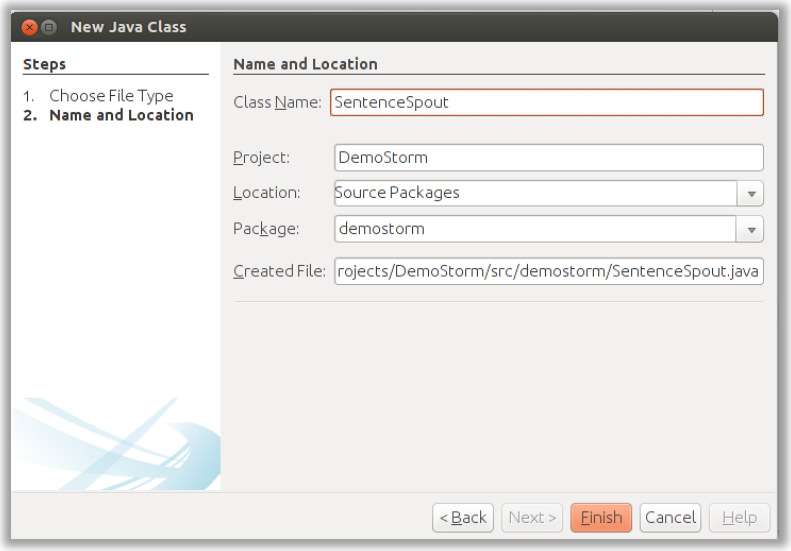

Figure 19. Ajout d'une classe Spout

Cette classe doit dériver de **BaseRichSpout** :

```
public class SentenceSpout extends BaseRichSpout{

    ......................

}
```

Une fois cet héritage ajouté, on peut utiliser les mécanismes automatiques de NetBeans, en cliquant sur le côté gauche de l'éditeur qui propose de créer une implémentation pour toutes les méthodes abstraites (figure 20).

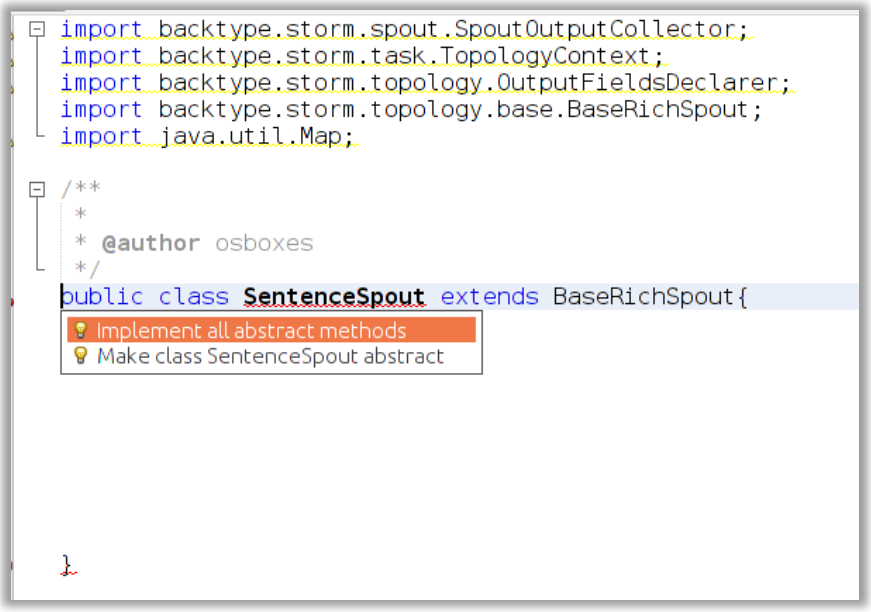

Figure 20. Ajout des méthodes abstraites

Les méthodes ajoutées sont précédées de l'annotation **@Override** et elles sont au nombre de trois comme le montre le code ci-dessous :

```
package demostorm;

import backtype.storm.spout.SpoutOutputCollector;
```

```
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import java.util.Map;

public class SentenceSpout extends BaseRichSpout{

  @Override
  public void declareOutputFields(OutputFieldsDeclarer ofd) {
   throw new UnsupportedOperationException("Not supported yet.");
   //To change body of generated methods, choose Tools | Templates.
  }

  @Override
  public void open(Map map, TopologyContext tc, SpoutOutputCollector soc) {
   throw new UnsupportedOperationException("Not supported yet.");
   //To change body of generated methods, choose Tools | Templates.
  }

  @Override
  public void nextTuple() {
   throw new UnsupportedOperationException("Not supported yet.");
   //To change body of generated methods, choose Tools | Templates.
  }
}
```

Quatre modifications de la classe sont nécessaires.

Modification 1 : ajout d'un poème dans la classe SentenceSpout et déclaration d'un flux

Dans la classe **SentenceSpout** on introduit un tableau (**liste_de_phrases**) contenant les phrases du poème de Verlaine et un index initialisé à 0. Le code est alors le suivant :

```
private String[] liste_de_phrases =
  {
  "A un passant",
  "Mon cher enfant que j'ai vu dans ma vie errante,",
  "Mon cher enfant, que, mon Dieu, tu me recueillis,",
  "Moi-même pauvre ainsi que toi, purs comme lys,",
  "Mon cher enfant que j'ai vu dans ma vie errante !",
  "Et beau comme notre âme pure et transparente,",
  "Mon cher enfant, grande vertu de moi, la rente,",
  "De mon effort de charité, nous, fleurs de lys !",
  "On te dit mort... Mort ou vivant, sois ma mémoire !",
  "Et qu'on ne hurle donc plus que c'est de la gloire",
  "Que je m'occupe, fou qu'il fallut et qu'il faut...",
  "Petit ! mort ou vivant, qui fis vibrer mes fibres,",
  "Quoi qu'en aient dit et dit tels imbéciles noirs",
  "Compagnon qui ressuscitas les saints espoirs,",
  "Va donc, vivant ou mort, dans les espaces libres !",
  "Paul Verlaine"
  };
private int index = 0;
```

Toujours dans la classe, on peut déclarer un flux de sortie de type **SpoutOutputCollector** comme suit :

```
private SpoutOutputCollector flux_sortie;
```

Modification 2 : Mise à jour de la méthode **declareOutputFields()**

Cette méthode crée un tuple dont la première partie comprend la chaine **sentence**. Cela se fait en définissant un champ de type **Fields** et en ajoutant ce champ au tuple de sortie.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer ofd) {
  Fields champ = new Fields("sentence");
  ofd.declare(champ);
}
```

Modification 3 : Mise à jour de la méthode **nextTuple()**

Cette méthode récupère la ligne courante dans le tableau **liste_de_phrases[…]** et créée la partie valeur du tuple (**new Values(phrase)**), puis émet ce tuple sur le flux de sortie avec la méthode **flux_sortie.emit(…)**. La fin de la procédure consiste à incrémenter **index** et à le remette à 0 si nécessaire. Afin d'éviter la génération trop rapide d'un flux de mot, une attente d'une milliseconde est ajoutée à la fin de la procédure.

Il est souhaitable que la procédure n'émette plus de phrase lorsque le poème aura été émis. Pour cela, il suffit d'ajouter une variable globale à la classe et de tester sa valeur. Si **termine** vaut 0, la liste de phrase n'est pas épuisée et dans le cas contraire, on se contente de réaliser l'attente d'une milliseconde.

```
@Override
public void nextTuple() {

  if (termine==0)
  {
    String phrase = liste_de_phrases[index];
    Values valeur = new Values(phrase);
    flux_sortie.emit(valeur);
    System.out.println("Spout : emission de "+phrase);

    index++;
    if (index>=liste_de_phrases.length)
    {
     index = 0;
     termine=1;
    }
  }
  Utils.waitForMillis(1);
 }
```

Cette variable doit être déclarée en statique dans la classe.

```
private static int termine = 0;
```

Dans ce cas, il est souhaitable d'ajouter une méthode pour consulter l'état de la variable :

```
public int consulter_etat() {
  int res = termine;
  return res;
}
```

Modification 4 : Mise à jour de la méthode **open()**

La méthode se contente de stocker dans la variable flux_sortie le flux de sortie passé en paramètre.

```
@Override
public void open(Map map, TopologyContext tc, SpoutOutputCollector soc) {
```

```
    this.flux_sortie = soc;
}
```

Le code complet de la classe est donné ci-dessous :

```
package demostorm;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import java.util.Map;

public class SentenceSpout extends BaseRichSpout{

    private String[] liste_de_phrases =
    {
    "A un passant",
    "Mon cher enfant que j'ai vu dans ma vie errante,",
    "Mon cher enfant, que, mon Dieu, tu me recueillis,",
    "Moi-même pauvre ainsi que toi, purs comme lys,",
    "Mon cher enfant que j'ai vu dans ma vie errante !",
    "Et beau comme notre âme pure et transparente,",
    "Mon cher enfant, grande vertu de moi, la rente,",
    "De mon effort de charité, nous, fleurs de lys !",
    "On te dit mort... Mort ou vivant, sois ma mémoire !",
    "Et qu'on ne hurle donc plus que c'est de la gloire",
    "Que je m'occupe, fou qu'il fallut et qu'il faut...",
    "Petit ! mort ou vivant, qui fis vibrer mes fibres,",
    "Quoi qu'en aient dit et dit tels imbéciles noirs",
    "Compagnon qui ressuscitas les saints espoirs,",
    "Va donc, vivant ou mort, dans les espaces libres !",
    "Paul Verlaine"
    };
    private int index = 0;
    private static int termine = 0;

    private SpoutOutputCollector flux_sortie;


    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {

        Fields champ = new Fields("sentence");
        ofd.declare(champ);

    }

    @Override
    public void open(Map map, TopologyContext tc, SpoutOutputCollector soc)
{
        this.flux_sortie = soc;
    }

    public int consulter_etat() {
      int res = termine;
      return res;
    }
```

```
   @Override
    public void nextTuple() {
      if (termine==0)
        {
        String phrase = liste_de_phrases[index];
        Values valeur = new Values(phrase);
        flux_sortie.emit(valeur);
        System.out.println("Spout : emission de "+phrase);
        index++;
        if (index>=liste_de_phrases.length)
           {
            index = 0;
            termine=1;
           }
        }
      Utils.waitForMillis(1);
    }
}
```

4. Création du Bolt chargé du découpage en mot

Le Bolt a pour objectif de créer un flux (stream) de couple <clé;valeur> de la forme

{ "word":"**xxxxxxxxxxxxxx**" }

où "**xxxxxxxxxxxxxx**" est un mot. On obtient par exemple un stream comme celui-ci :

{ "word":"Mon" }, { "word":"cher" }, { "word":"enfant" }

Ceci correspond à la deuxième fonctionnalité représentée sur le schéma de la figure 21.

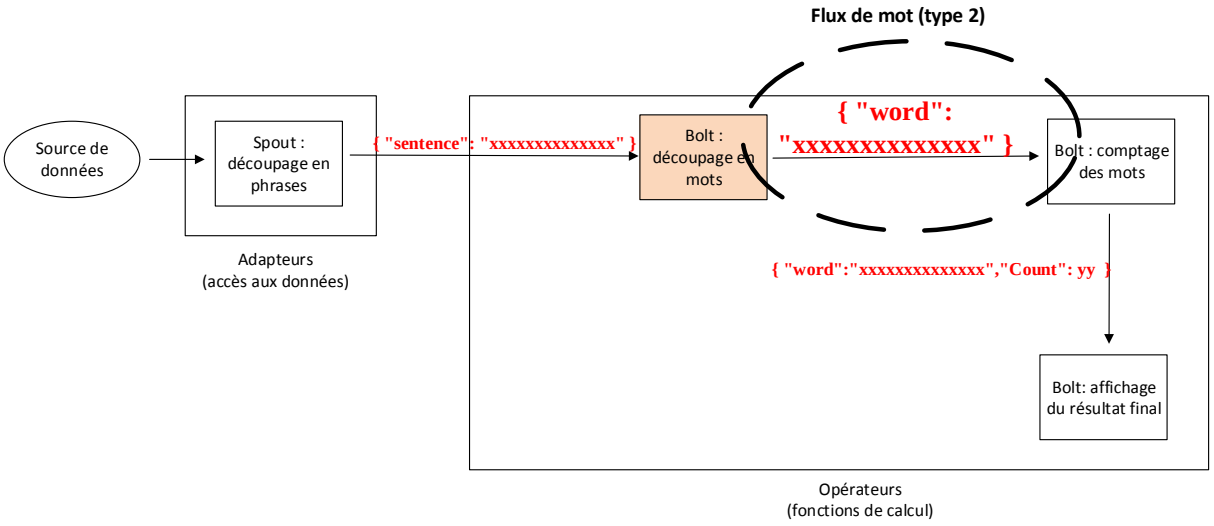

Figure 21. Création du Bolt pour le découpage en mots

Il faut ajouter une classe SplitSentenceBolt au projet (figure xxx).

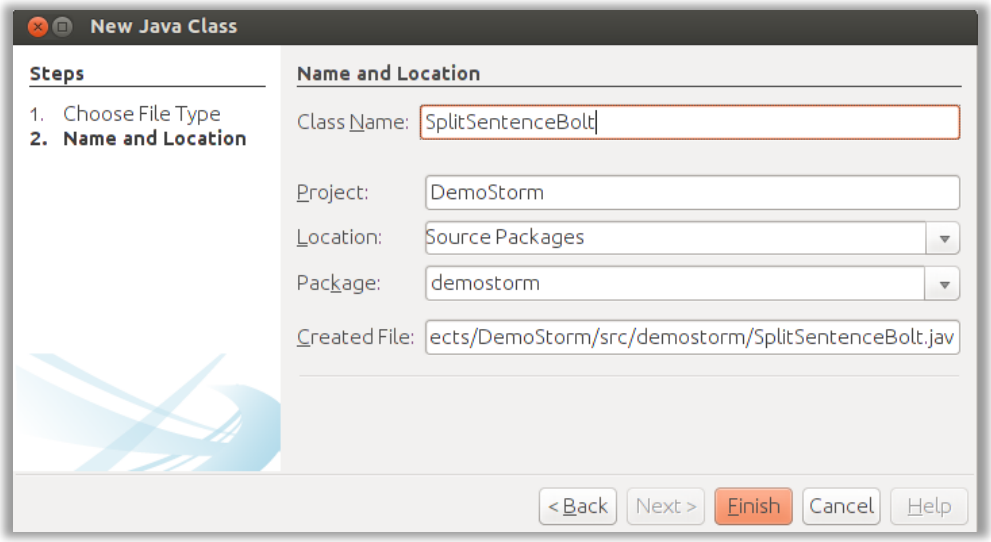

Figure 22. Définition d'une classe **SplitSentenceBolt**

Comme pour le Spout il faut ajouter une relation d'héritage avec cette fois **BaseRichBolt** et utiliser NetBeans pour la génération automatique du code correspondant aux méthodes "abstract" de la superclasse (figure 23).

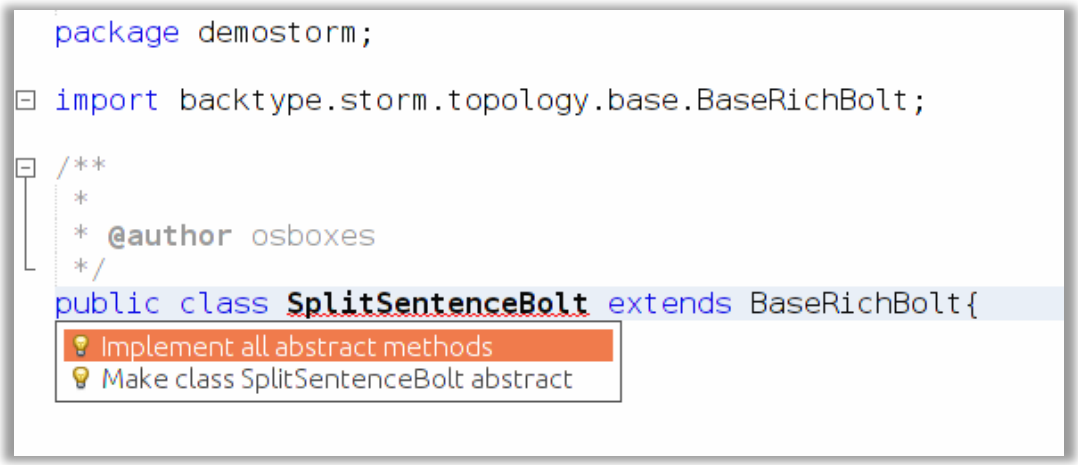

Figure 23. Ajout d'une relation d'héritage

Le code de la classe est alors un code identique à celui-ci et comprend 3 méthodes :

```
package demostorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Tuple;
import java.util.Map;

public class SplitSentenceBolt extends BaseRichBolt{

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
```

```
    public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void execute(Tuple tuple) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

Quatre modifications de la classe sont nécessaires.

Modification 1 : ajout d'un flux de sortie

Dans la classe on ajoute une variable flux_entree qui va servir à récupérer le flux en provenance du Spout.

```
private OutputCollector flux_sortie;
```

Modification 2 : mise à jour de la méthode **prepare()**.

Cette méthode stocke dans la variable flux_entree, le flux reçu en paramètre.

```
@Override
public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
   this.flux_entree = oc;
}
```

Modification 3 : mise à jour de la méthode **execute()**.

Cette méthode reçoit un paramètre de la forme <"sentence":"xxxxx">. La première étape consiste à récupérer le champ valeur du tuple. Ceci se fait par l'utilisation de la méthode **getStringByField** :

```
String la_phrase_recue = tuple.getStringByField("sentence");
```

On découpe ensuite la phrase en utilisant les guillemets comme séparateur et on obtient un tableau de mots.

```
String[] liste_de_mots = la_phrase_recue.split(" ");
```

Pour finir, on parcourt le tableau de mots et pour chaque mot, on envoie le mot sur le flux de sortie.

```
 for(String mot : liste_de_mots){
     Values une_valeur =  new Values(mot);
     this.flux_sortie.emit(une_valeur);
  }
```

Le code de la méthode est le suivant :

```
@Override
public void execute(Tuple tuple) {
 String la_phrase_recue = tuple.getStringByField("sentence");
 String[] liste_de_mots = la_phrase_recue.split(" ");
 for(String mot : liste_de_mots){
     Values une_valeur =  new Values(mot);
     this.flux_sortie.emit(une_valeur);
  }
}
```

Modification 4 : mise à jour de la méthode **declareOutputFields()**.

Cette méthode crée un tuple dont la première partie comprend la chaine **word**. Cela se fait en définissant un champ de type **Fields** et en ajoutant ce champ au tuple de sortie.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer ofd) {
  Fields champ = new Fields("word");
  ofd.declare(champ);
}
```

Le code de la classe est alors un code identique à celui-ci et comprend 3 méthodes :

```
package demostorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import java.util.Map;


public class SplitSentenceBolt extends BaseRichBolt{

    private OutputCollector flux_sortie;

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        Fields champ = new Fields("word");
        ofd.declare(champ);
    }

    @Override
    public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
        this.flux_sortie = oc;
    }

    @Override
    public void execute(Tuple tuple) {
        String la_phrase_recue = tuple.getStringByField("sentence");
        String[] liste_de_mots = la_phrase_recue.split(" ");
        for(String mot : liste_de_mots){
           Values une_valeur =  new Values(mot);
           this.flux_sortie.emit(une_valeur);
        }
    }

}
```

5. Création du Bolt chargé du décompte des mots

Le Bolt a pour objectif de créer un flux (stream) de couples <clé;valeur> de la forme

{ "word":"**xxxxxxxxxxxxxx**" , "Count":yy}

où "**xxxxxxxxxxxxxx**" est un mot et yy un nombre entier représentant le nombre d'occurrences du mot "**xxxxxxxxxxxxxx**". On obtient par exemple un stream comme celui-ci :

{ "word":"vie ", "Count":5 } qui signifie que le mot "vie" apparaît 5 fois

Ceci correspond à la troisième fonctionnalité représentée sur le schéma de la figure 24.

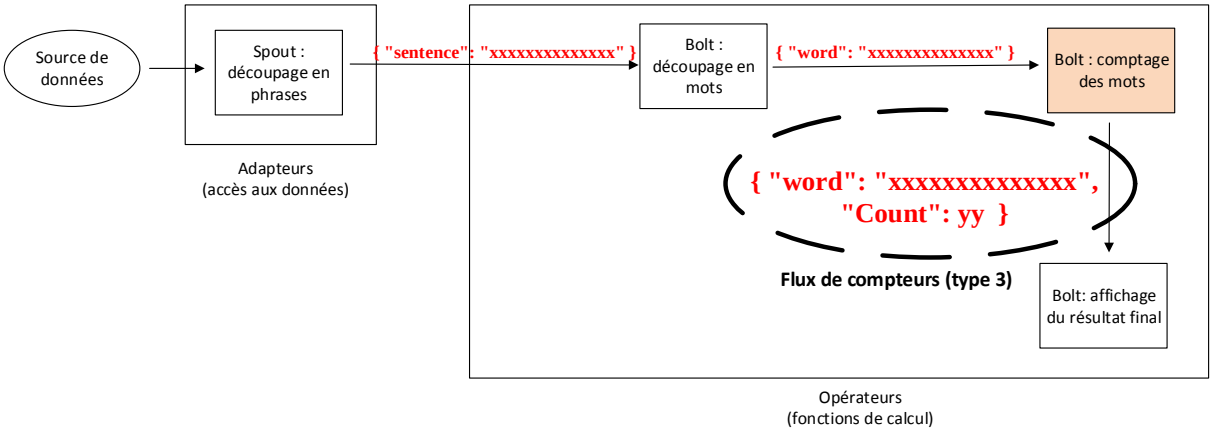

Figure 24. Création du Bolt pour le décompte des mots

Il faut ajouter une nouvelle classe au projet. Cette classe est nommée : **WordCoutBolt** (figure 25).

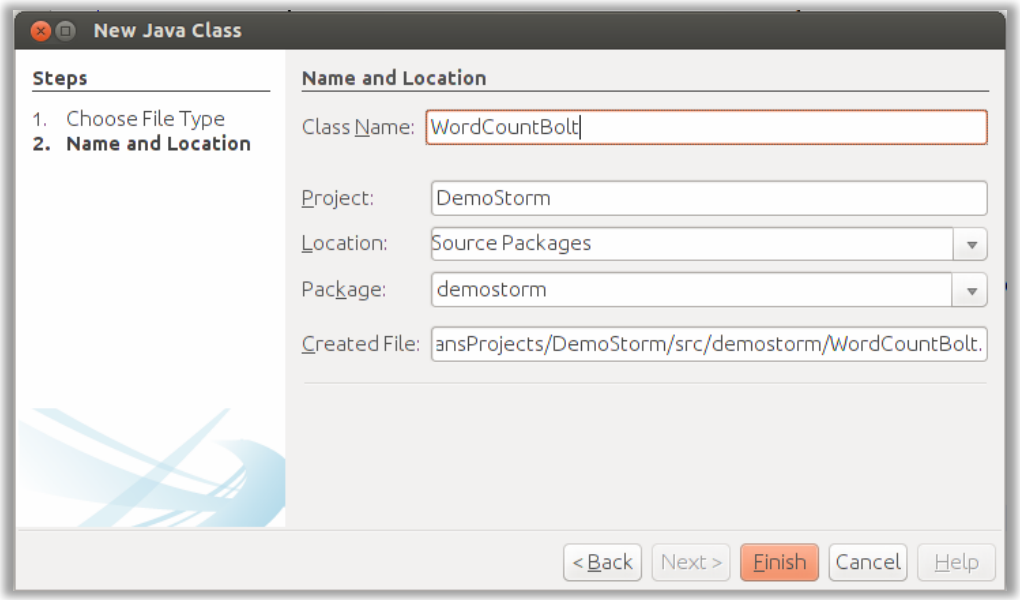

Figure 25. Création d'une classe **WordCountBolt**

La classe doit hériter de **BaseRichBolt** et implémenter les 3 méthodes abstraites de la superclasse. Le code est alors celui-ci :

```
package demostorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
```

```
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Tuple;
import java.util.Map;


public class WordCountBolt extends BaseRichBolt{

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void execute(Tuple tuple) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

Quatre modifications de la classe sont nécessaires.

Modification 1 : ajout d'un flux de sortie et d'une zone de stockage

Dans la classe on ajoute une variable flux_sortie qui va servir à créer le flux et une **HashMap** pour stocker les informations du type **<String, Long>**.

```
private OutputCollector flux_sortie;
private HashMap<String, Long> compteurs = null;
```

Modification 2 : mise à jour de la méthode **prepare()**.

La variable **flux_sortie** est initialisée avec **oc** passée en paramètre et **compteurs** est initialisée par un appel au constructeur de la classe **HashMap**.

```
@Override
public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
   this.flux_sortie = oc;
   this.compteurs = new HashMap<String, Long>();
}
```

Modification 3 : mise à jour de la méthode **declareOutputFields**().

Cette méthode crée le flux de sortie qui est du type :

{ "word":"**xxxxxxxxxxxxxx**" , "Count":yy}

Deux champs de données sont ajoutés : "word" et "count".

Le code de la méthode est :

```
@Override
public void declareOutputFields(OutputFieldsDeclarer ofd) {
    Fields champ = new Fields("word", "Count");
    ofd.declare(champ);
}
```

Modification 4 : mise à jour de la méthode **execute**().

En consultant le champ "word" on récupère en premier lieu le mot concerné qu'on stocke dans une variable **le_mot** de type String.

```
String le_mot = tuple.getStringByField("word");
```

On peut ensuite consulter le **HashMap** en donnant comme clé de recherche le_mot :

```
Long valeur_courante = this.compteurs.get(le_mot);
```

Deux cas sont alors à prendre en compte :

- soit la valeur du compteur **valeur_courante** est **null** et cela signifie que le mot n'a encore jamais été trouvé et qu'il ne figure pas dans la map ;

- soit la valeur du compteur est différente de **null** et **valeur_courante** dans ce cas, contient le nombre d'occurrences du mot **le_mot** qui ont déjà été comptabilisées.

Dans le premier cas, il faut initialiser valeur_courante à 0 et dans l'autre cas, on va incrémenter valeur_courante. Ceci donne le code Java suivant :

```
if(valeur_courante == null){
  valeur_courante = 0L;
}
valeur_courante++;
```

Pour finir, il faut renseigner les deux champs du flux de sortie en créant une variable du type Values (qu'il faut renseigner avec le_mot et valeur_courante) puis, émettre la donnée sur le flux de sortie.

```
Values une_valeur =  new Values(le_mot, valeur_courante);
this.flux_sortie.emit(une_valeur);
```

Le code complet de la méthode est le suivant :

```
@Override
public void execute(Tuple tuple) {
  String le_mot = tuple.getStringByField("word");
  Long valeur_courante = this.compteurs.get(le_mot);
  if(valeur_courante == null){
     valeur_courante = 0L;
  }
  valeur_courante++;
  this.compteurs.put(le_mot, valeur_courante);

  Values une_valeur =  new Values(le_mot, valeur_courante);
  this.flux_sortie.emit(une_valeur);

}
```

6. Création du Bolt chargé de l'affichage final

Le Bolt a pour objectif de créer les rapports de sorties c'est-à-dire d'afficher le nombre d'occurrences de chaque mot dans la source (ici un poème de Verlaine). Il s'agit du dernier élément de la topologie (figure 26).

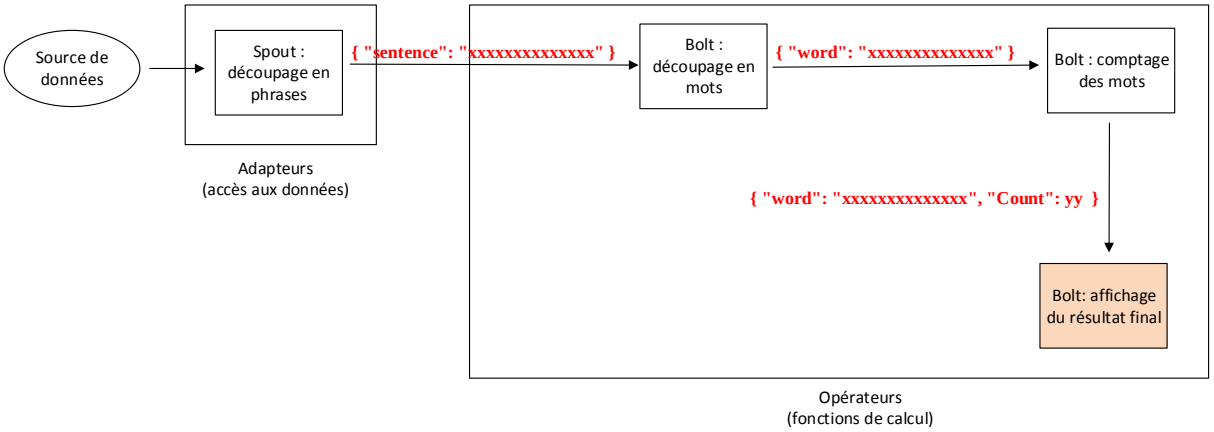

Figure 26. Création du Bolt pour l'affichage du résultat final

Il faut ajouter une nouvelle classe dans le projet, nommée par exemple ReportBolt, et faire hériter cette classe de BaseRichBolt et proposer une implémentation des méthodes abstraites.

Ceci donne une première version de la classe similaire à celle-ci :

```
package demostorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Tuple;
import java.util.Map;

public class ReportBolt extends BaseRichBolt {

    @Override
    public void declareOutputFields(OutputFieldsDeclarer ofd) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void execute(Tuple tuple) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

Cinq modifications de la classe sont nécessaires.

Modification 1 : ajout d'un compteur de type HashMap

Dans la classe on ajoute une variable **compteur_final** qui va servir à stocker les informations du type **<String, Long>** c'est-à-dire à stocker le nombre d'occurrences de chaque mot.

```
private HashMap<String, Long> compteur_final = null;
```

Modification 2 : mise à jour de la méthode **prepare()**.

Cette méthode initialise le tableau **compteur_final**.

```
@Override
public void prepare(Map map, TopologyContext tc, OutputCollector oc) {
    this.compteur_final = new HashMap<String, Long>();
}
```

Modification 3 : mise à jour de la méthode **declareOutputFields()**.

L'unité de calcul Bolt ne va créer aucun flux de sortie. Cette méthode reste donc vide et aucun code n'est à ajouter dans la méthode.

```
@Override
public void declareOutputFields(OutputFieldsDeclarer ofd) {

}
```

Modification 4 : mise à jour de la méthode **execute()**.

La méthode récupère la valeur du champ word (on accède ont au mot stocké) et la valeur du champ count (nombre d'occurrences du mot) et stocke ensuite dans le tableau compteur_final, chaque couple **<mot, nb d'occurrences>**.

```
@Override
public void execute(Tuple tuple) {
  String word = tuple.getStringByField("word");
  Long nb_de_mots = tuple.getLongByField("count");
  this.compteur_final.put(word, nb_de_mots);
}
```

Modification 5 : Création d'une nouvelle méthode **cleanup()**

La méthode **cleanup()** parcours l'ensemble des mots stockés dans le tableau **compteur_final**.

Pour cela on passe par une Liste (variable keys) qu'on initialise avec les clés du compteur_final. Cela se fait en 2 lignes de code :

```
List<String> keys = new ArrayList<String>();
keys.addAll(this.compteur_final.keySet());
```

L'ensemble de ces clés (ici il s'agit de mots) sont triées avec la méthode **sort**.

```
Collections.sort(keys);
```

Pour finir, une boucle for permet de parcourir l'ensemble de toutes les clés (ici des mots) et de récupérer la valeur entières correspondantes dans le tableau **compteur_final**.

Le code de la méthode est le suivant :

```
public void cleanup() {
  System.out.println("--- Nb d'occurences ---");
  List<String> keys = new ArrayList<String>();
  keys.addAll(this.compteur_final.keySet());
  Collections.sort(keys);
  for (String key : keys) {
     System.out.println(key + " : " + this.compteur_final.get(key));
  }
  System.out.println("--------------");
}
```

7. Création de la topologie (programme principal)

Il s'agit de créer une topologie (c'est-à-dire une architecture) se composant d'instances des classes précédemment définies selon le schéma de la figure 27. Pour des raisons de simplicité la topologie se nomme "Topology : compter les mots".

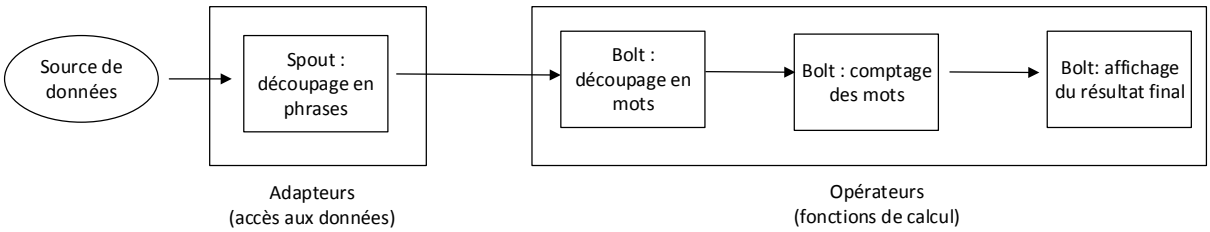

Figure 27. Topologie "Topology : compter les mots"

Partie 1. Création de constantes

On définit 5 constantes dont une pour la topologie. Ces constantes sont des chaînes de caractères utilisées par la suite pour définir les instances des classes.

```
private static final String SPOUT_ID = "Spout-generateur-de-phrases";
private static final String BOLT_DECOUPE_ID = "bolt-decoupage-en-mots";
private static final String BOLT_COMPTEUR_ID = "bolt-compteur-de-mots";
private static final String BOLT_AFFICHAGE_ID = "bolt-affichage";
private static final String TOPOLOGY_NOM = "Topology-compter-les-mots";
```

Attention, les chaînes de caractères ne doivent contenir aucun séparateur tel que, ou ; et l'espace est aussi interdit. Ainsi "Spout : generateur de phrases" est un choix à éviter car il provoquera des erreurs d'exécution par la suite.

Partie 2. Définition des variables

On définit 5 variables dont une pour la topologie. Ces variables sont des instances des classes précédemment crées. Cela donne le code suivant :

```
SentenceSpout spout_genere_phrases = new SentenceSpout();
SplitSentenceBolt bolt_decoupe = new SplitSentenceBolt();
WordCountBolt bolt_compte = new WordCountBolt();
ReportBolt bolt_affiche = new ReportBolt();
TopologyBuilder topologie = new TopologyBuilder();
```

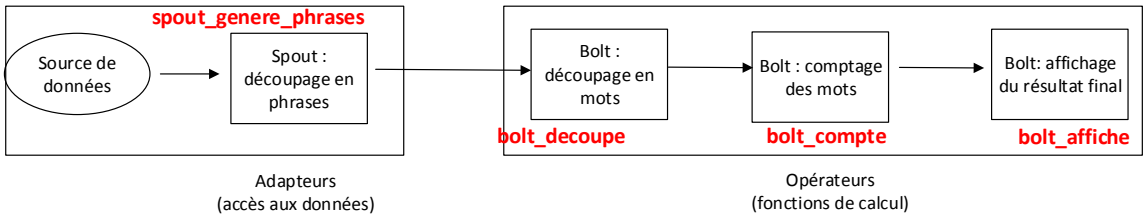

Figure 28. Les différents éléments de la Topologie

Ces unités de traitement sont ajoutées une à une à la topologie (figure 28) et doivent être liées à flux d'entrée à l'exception du Spout qui crée les données à traiter.

Le Spout

Pour le Spout il suffit d'utiliser la méthode **setSpout()** et de passer en paramètre l'identificateur du Spout (une chaîne de caractères nommée ici **SPOUT_ID**) et la variable de type **SentenceSpout.**

```
topologie.setSpout(SPOUT_ID, spout_genere_phrases);
```

Ceci donne la configuration de la figure 29 où seul le Spout apparaît.

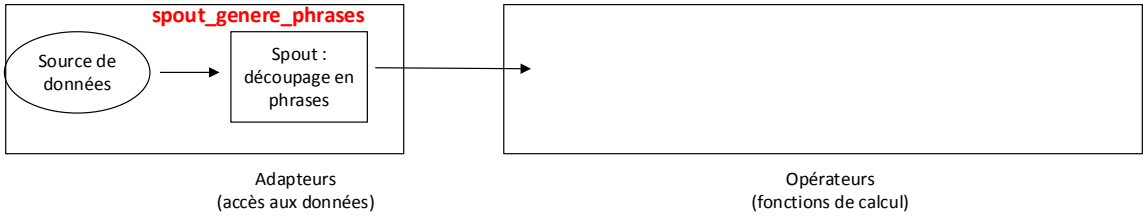

Figure 29. La topologie après l'ajout du Spout

Le Bolt de découpage des mots

Le Bolt chargé du découpage est ajouté à la topologie et lié au flux de sortie du Spout. Cela se fait par l'utilisation de la méthode **shuffleGrouping()** qui reçoit en paramètre le nom du Spout concerné. Cela veut dire que l'ensemble du flux crée par le Spout est redirigé vers le Bolt de découpage comme le montre la figure 30.

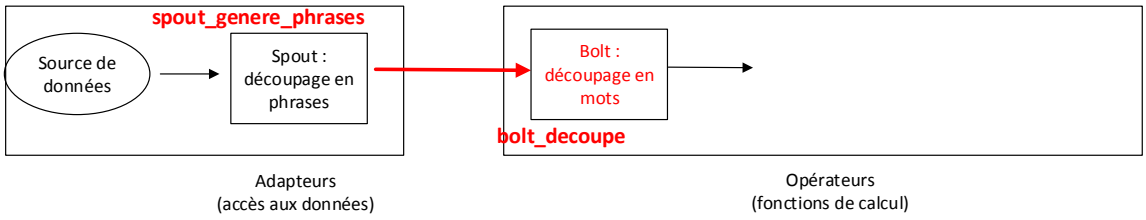

Figure 30. Ajout du Bolt de découpage lié au Spout

Le code Java est le suivant :

```
topologie.setBolt(BOLT_DECOUPE_ID, bolt_decoupe).shuffleGrouping(SPOUT_ID);
```

Le Bolt de comptage des mots

Le Bolt chargé du décompte des mots est ajouté à la topologie et lié au flux de sortie du Bolt de découpage. Cela se fait par l'utilisation de la méthode **FieldGrouping()** qui reçoit en paramètre (figure 31) :

- le nom du Bolt qui émet le flux qu'on reçoit en entrée (ici le Bolt de découpage) concerné.

- le nom d'un champ qui sert à regrouper les flux (ici le champ Word).

Le code Java est le suivant :

```
topologie.setBolt(BOLT_COMPTEUR_ID, bolt_compte).
        fieldsGrouping(BOLT_DECOUPE_ID, new Fields("word"));
```

Cela signifie que **tous les flux ayant la même valeur de champ Word seront redirigés vers le même Bolt de comptage**.

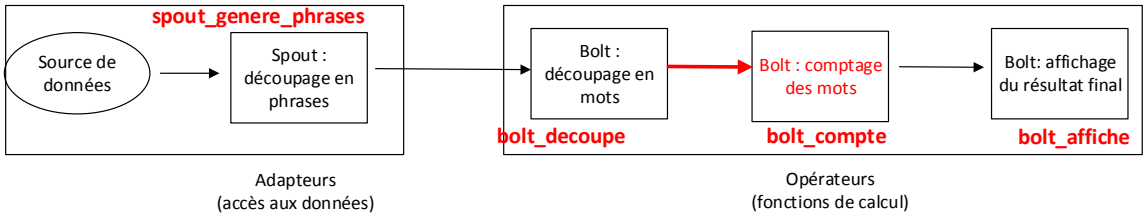

Figure 31. Ajout du Bolt de découpage lié au Spout

Le Bolt d'affichage des mots

Le Bolt chargé de l'affichage des mots est ajouté à la topologie et lié au flux de sortie du Bolt de comptage. Cela se fait par l'utilisation de la méthode **GlobalGrouping()** qui va permettre d'imposer que tous les flux émis par le Bolt compteur soient redirigés vers le même Bolt d'affichage (figure 32).

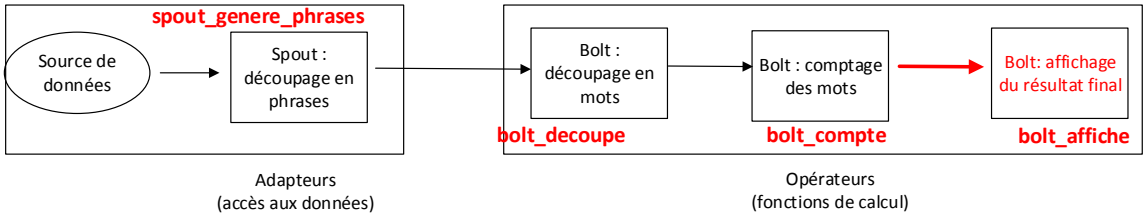

Figure 32. Ajout du Bolt de découpage lié au Spout

Le code Java est le suivant :

```
topologie.setBolt(BOLT_AFFICHAGE_ID,bolt_affiche).globalGrouping(BOLT_COMPTEUR_ID);
```

Le code Java correspondant à la définition des 3 bolts et du Spout est le suivant :

```
topologie.setSpout(SPOUT_ID, spout_genere_phrases);
topologie.setBolt(BOLT_DECOUPE_ID, bolt_decoupe).shuffleGrouping(SPOUT_ID);
topologie.setBolt(BOLT_COMPTEUR_ID, bolt_compte).
            fieldsGrouping(BOLT_DECOUPE_ID, new Fields("word"));
topologie.setBolt(BOLT_AFFICHAGE_ID,bolt_affiche).globalGrouping(BOLT_COMPTEUR_ID);
```

Partie 3. Démarrage de la topologie

La topologie est envoyée à Storm pour exécution en passant par la définition d'une configuration, d'un cluster, pour se terminer par un appel à **submitTopology()**.

```
Config configuration = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology(TOPOLOGY_NOM, configuration, topologie.createTopology());
```

Ensuite, on peut imaginer attendre la fin de la génération des phrases à analyser en consultant l'état du Spout toute les secondes pas exemple. Ceci donne alors une boucle de la forme :

```
int i=1;
while (spout_genere_phrases.consulter_etat()==0)
 {
    Utils.waitForSeconds(1);
 }
```

On peut ensuite par précaution, attendre 1 seconde avant de terminer la topologie et donc ajouter un waitForSecond()

```
Utils.waitForSeconds(1);
```

Il suffit ensuite, d'arrêter la topologie sur le cluster. Notons que l'appel à cluster.shutdown() va entraîner l'appel à la méthode clean() du Bolt d'affichage.

```
cluster.killTopology(TOPOLOGY_NOM);
cluster.shutdown();
```

Le code complet de la classe DemoStorm est le suivant :

```
package demostorm;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import static java.lang.Thread.yield;

public class DemoStorm {

    private static final String SPOUT_ID = "Spout-generateur-de-phrases";
    private static final String BOLT_DECOUPE_ID = "bolt-decoupage-en-mots";
    private static final String BOLT_COMPTEUR_ID = "bolt-compteur-de-mots";
    private static final String BOLT_AFFICHAGE_ID = "bolt-affichage";
    private static final String TOPOLOGY_NOM = "Topology-compter-les-mots";


    public static void main(String[] args) {

        SentenceSpout spout_genere_phrases = new SentenceSpout();
        SplitSentenceBolt bolt_decoupe = new SplitSentenceBolt();
        WordCountBolt bolt_compte = new WordCountBolt();
        ReportBolt bolt_affiche = new ReportBolt();
        TopologyBuilder topologie = new TopologyBuilder();

        topologie.setSpout(SPOUT_ID, spout_genere_phrases);
        topologie.setBolt(BOLT_DECOUPE_ID, bolt_decoupe).shuffleGrouping(SPOUT_ID);
        topologie.setBolt(BOLT_COMPTEUR_ID,
          bolt_compte).fieldsGrouping(BOLT_DECOUPE_ID, new Fields("word"));
        topologie.setBolt(BOLT_AFFICHAGE_ID,
           bolt_affiche).globalGrouping(BOLT_COMPTEUR_ID);


        Config configuration = new Config();
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology(TOPOLOGY_NOM, configuration,
           topologie.createTopology());


        int i=1;
        while (spout_genere_phrases.consulter_etat()==0)
        {
           Utils.waitForSeconds(1);

        }
        Utils.waitForSeconds(1);

        cluster.killTopology(TOPOLOGY_NOM);
        cluster.shutdown();
    }

}
```

7. Test et analyse de fonctionnement

Pour illustrer le principe de fonctionnement de la topologie, le contenu du poème a été limité aux trois premières phrases et le tableau dans la classe SentenceSpout est identique à celui-ci :

```
private String[] liste_de_phrases =
 {
```

```
 "a un passant",
 "mon cher enfant que j'ai vu dans ma vie errante",
 "mon cher enfant que mon Dieu, tu me recueillis"
};
```

Remarquons que nous avons supprimé les majuscules au début de phrase de sorte que le mot "mon" apparaisse deux fois sur la ligne numéro 3.

D'autre part, nous avons ajouté des affichages dans les méthodes suivantes en prenant soin de taguer tous les messages avec l'heure courante.

1) **nextTuple()** de la classe **SentenceSpout**;

```
@Override
public void nextTuple() {
  if (termine==0)
  {
    String phrase = liste_de_phrases[index];
    Values valeur = new Values(phrase);

    Calendar cal = Calendar.getInstance();
    int hour = cal.get(Calendar.HOUR_OF_DAY);
    int minute = cal.get(Calendar.MINUTE);
    int secondes= cal.get(Calendar.SECOND);
    int msecondes= cal.get(Calendar.MILLISECOND);
    String heure = hour+":"+minute+":"+secondes+":"+msecondes;

    System.out.println(""+heure+" : Spout : emission de "+phrase);

    flux_sortie.emit(valeur);
    index++;
    if (index>=liste_de_phrases.length)
    {
      index = 0;
      termine=1;
    }
  }
  Utils.waitForMillis(1);
}
```

Attention à insérer ces affichages avant l'appel à la méthode **emit()** faute de quoi, l'heure afficher ne sera pas représentative du moment de l'appel mais uniquement de la fin d'exécution de la méthode **emit()**.

2) **execute()** de la classe **SplitSentenceBolt**;

```
@Override
public void execute(Tuple tuple) {
   String la_phrase_recue = tuple.getStringByField("sentence");

   Calendar cal = Calendar.getInstance();
   int hour = cal.get(Calendar.HOUR_OF_DAY);
   int minute = cal.get(Calendar.MINUTE);
   int secondes= cal.get(Calendar.SECOND);
   int msecondes= cal.get(Calendar.MILLISECOND);
   String heure = hour+":"+minute+":"+secondes+":"+msecondes;

   System.out.println(""+heure+"Bolt : reception de "+la_phrase_recue);

   String[] liste_de_mots = la_phrase_recue.split(" ");
   for(String mot : liste_de_mots){
      Values une_valeur =  new Values(mot);
```

```
      cal = Calendar.getInstance();
      hour = cal.get(Calendar.HOUR_OF_DAY);
      minute = cal.get(Calendar.MINUTE);
      secondes= cal.get(Calendar.SECOND);
      msecondes= cal.get(Calendar.MILLISECOND);
      heure = hour+":"+minute+":"+secondes+":"+msecondes;

      System.out.println(""+heure+"Bolt : emission de "+mot);

      this.flux_sortie.emit(une_valeur);

    }
}
```

3) **execute()** de la classe **WordCountBolt**;

```
@Override
public void execute(Tuple tuple) {
   String le_mot = tuple.getStringByField("word");
   Long valeur_courante = this.compteurs.get(le_mot);
   if(valeur_courante == null){
       valeur_courante = 0L;
   }
   valeur_courante++;

   Calendar cal = Calendar.getInstance();
   int hour = cal.get(Calendar.HOUR_OF_DAY);
   int minute = cal.get(Calendar.MINUTE);
   int secondes= cal.get(Calendar.SECOND);
   int msecondes= cal.get(Calendar.MILLISECOND);
   String heure = hour+":"+minute+":"+secondes+":"+msecondes;

   System.out.println(""+heure+"Bolt compute --> "+le_mot +
          " : " + valeur_courante);

   this.compteurs.put(le_mot, valeur_courante);

   Values une_valeur =  new Values(le_mot, valeur_courante);
   this.flux_sortie.emit(une_valeur);
}
```

4) **execute()** de la classe **ReportBolt**;

```
@Override
public void execute(Tuple tuple) {
  String word = tuple.getStringByField("word");
  Long nb_de_mots = tuple.getLongByField("count");
  this.compteur_final.put(word, nb_de_mots);
  Calendar cal = Calendar.getInstance();
  int hour = cal.get(Calendar.HOUR_OF_DAY);
  int minute = cal.get(Calendar.MINUTE);
  int secondes= cal.get(Calendar.SECOND);
  int msecondes= cal.get(Calendar.MILLISECOND);
  String heure = hour+":"+minute+":"+secondes+":"+msecondes;

  System.out.println(""+heure+"Bold Final --> "+word+" : "+nb_de_mots);
}
```

Traces d'exécution :

```
15:41:22:610 Spout : emission de A un passant
15:41:22:614 Bolt : reception de A un passant
15:41:22:614 Bolt : emission de A
15:41:22:614 Bolt : emission de un
15:41:22:614 Bolt : emission de passant
```

```
15:41:22:615 Bolt compute --> A : 1
15:41:22:616 Bold Final --> A : 1
15:41:22:618 Bolt compute --> un : 1
15:41:22:618 Bold Final --> un : 1
15:41:22:618 Bolt compute --> passant : 1
15:41:22:618 Bold Final --> passant : 1
15:41:22:621 : Spout : emission de mon cher enfant que j'ai vu dans ma vie errante
15:41:22:621 Bolt : reception de mon cher enfant que j'ai vu dans ma vie errante
15:41:22:621 Bolt : emission de mon
15:41:22:621 Bolt : emission de cher
15:41:22:621 Bolt : emission de enfant
15:41:22:621 Bolt : emission de que
15:41:22:621 Bolt : emission de j'ai
15:41:22:621 Bolt : emission de vu
15:41:22:621 Bolt : emission de dans
15:41:22:621 Bolt : emission de ma
15:41:22:621 Bolt : emission de vie
15:41:22:621 Bolt : emission de errante
15:41:22:623 Bolt compute --> mon : 1
15:41:22:624 Bold Final --> mon : 1
15:41:22:624 Bolt compute --> cher : 1
15:41:22:624 Bold Final --> cher : 1
15:41:22:624 Bolt compute --> enfant : 1
15:41:22:624 Bold Final --> enfant : 1
15:41:22:624 Bolt compute --> que : 1
15:41:22:624 Bold Final --> que : 1
15:41:22:624 Bolt compute --> j'ai : 1
15:41:22:624 Bold Final --> j'ai : 1
15:41:22:624 Bolt compute --> vu : 1
15:41:22:625 Bold Final --> vu : 1
15:41:22:627 : Spout : emission de mon cher enfant que mon Dieu, tu me recueillis
15:41:22:627 Bolt : reception de mon cher enfant que mon Dieu, tu me recueillis
15:41:22:628 Bolt : emission de mon
15:41:22:628 Bolt : emission de cher
15:41:22:628 Bolt : emission de enfant
15:41:22:628 Bolt : emission de que
15:41:22:628 Bolt : emission de mon
15:41:22:628 Bolt : emission de Dieu,
15:41:22:628 Bolt : emission de tu
15:41:22:629 Bolt : emission de me
15:41:22:629 Bolt : emission de recueillis
15:41:22:629 Bolt compute --> dans : 1
15:41:22:629 Bold Final --> dans : 1
15:41:22:629 Bolt compute --> ma : 1
15:41:22:630 Bold Final --> ma : 1
15:41:22:630 Bolt compute --> vie : 1
15:41:22:630 Bold Final --> vie : 1
15:41:22:630 Bolt compute --> errante : 1
15:41:22:631 Bold Final --> errante : 1
15:41:22:631 Bolt compute --> mon : 2
15:41:22:631 Bold Final --> mon : 2
15:41:22:631 Bolt compute --> cher : 2
15:41:22:632 Bold Final --> cher : 2
15:41:22:632 Bolt compute --> enfant : 2
15:41:22:632 Bold Final --> enfant : 2
15:41:22:632 Bolt compute --> que : 2
15:41:22:632 Bold Final --> que : 2
15:41:22:633 Bolt compute --> mon : 3
15:41:22:633 Bold Final --> mon : 3
15:41:22:633 Bolt compute --> Dieu, : 1
15:41:22:633 Bold Final --> Dieu, : 1
15:41:22:634 Bolt compute --> tu : 1
15:41:22:634 Bold Final --> tu : 1
15:41:22:634 Bolt compute --> me : 1
15:41:22:635 Bold Final --> me : 1
15:41:22:635 Bolt compute --> recueillis : 1
15:41:22:639 Bold Final --> recueillis : 1
```

8. Le résultat final

```
--- Nb d'occurences ---
A : 1
Dieu, : 1
cher : 2
dans : 1
enfant : 2
errante : 1
j'ai : 1
ma : 1
me : 1
mon : 3
passant : 1
que : 2
recueillis : 1
tu : 1
un : 1
vie : 1
vu : 1
```

9. Utilisation sur un cluster Storm existant

Le code Java fourni lance une exécution de storm locale. Il n'est pas possible en l'état de faire exécuter ce code sur un serveur Storm en utilisant les ressources du serveur.

Exemple.

Il faut se rendre dans le répertoire **bin** de **storm** et taper en ligne de commande :

```
storm jar DemoStorm.jar demostorm.DemoStorm DemoStorm demostorm.DemoStorm DemoStorm
-c nimbus.host=localhost
```

Ceci donne le résultat de la figure 33.

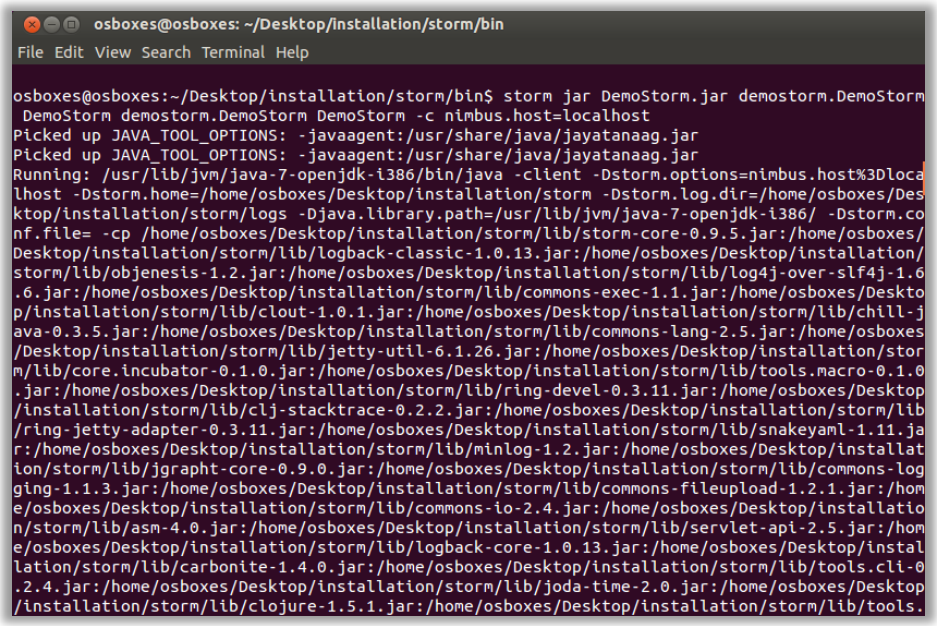

Figure 33. Exécution du code Java dans un cluster Storm

Le résultat est celui de la figure 34.

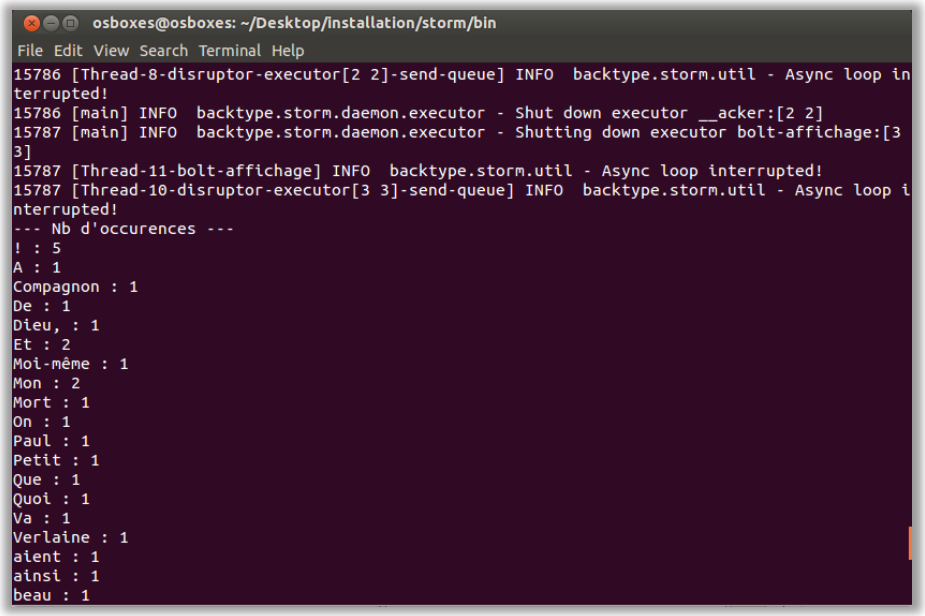

Figure 34. Résultat d'exécution du code Java dans un cluster Storm

Ce résultat est identique à celui obtenu en lançant le code à partir de NetBeans. Si on lance Firefox et qu'on consulte la page http://localhost:8772 on constatera qu'aucune topologie n'est active.

Modification du code

On va distinguer le cas où le programme s'exécute en local (on ne lui passe alors aucun paramètre) et le cas où il s'exécute sur le cluster Storm.
L'exécution sur le cluster Storm se fait par un appel à la méthode **submitTopology** qui reçoit comme premier paramètre le nom de la topologie sur le cluster.

Le code est le suivant :

```
if (args != null && args.length > 0) {
 StormSubmitter.submitTopology(args[0], configuration, topologie.createTopology());
}
else
  {

  LocalCluster cluster = new LocalCluster();

  cluster.submitTopology(TOPOLOGY_NOM, configuration, topologie.createTopology());

  int i=1;
  while (spout_genere_phrases.consulter_etat()==0)
  {
   Utils.waitForSeconds(1);
  }
  Utils.waitForSeconds(20);

  cluster.killTopology(TOPOLOGY_NOM);
  cluster.shutdown();

}
```

Il faut se rendre dans le répertoire **bin** de **storm** et taper en ligne de commande :

```
storm jar DemoStorm.jar demostorm.DemoStorm DemoStorm demostorm.DemoStorm DemoStorm
-c nimbus.host=localhost
```

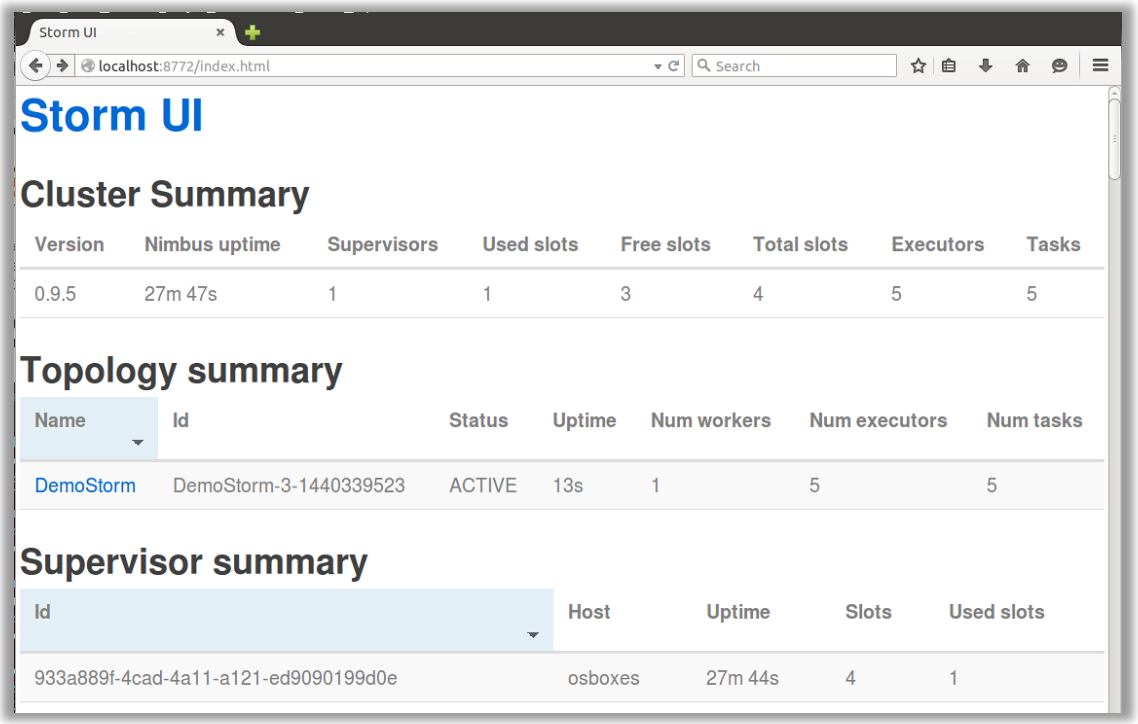

Figure 35. La topologie en cours d'exécution

En cliquant sur le nom de la topologie, on peut accéder à des informations précises (figure 36)

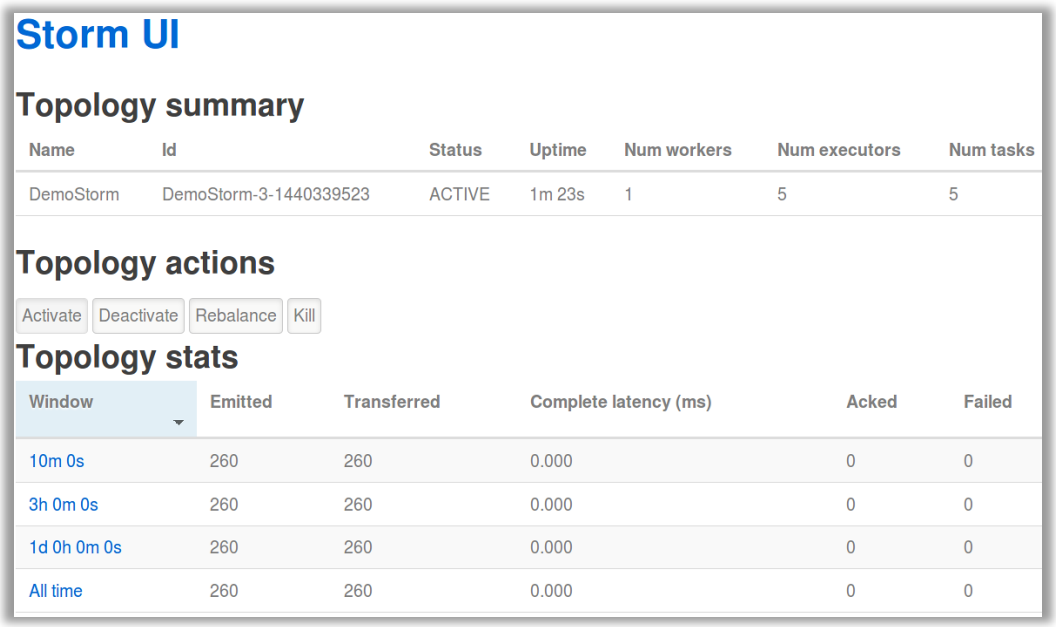

Figure 36. Détails sur la topologie

Evidemment on retrouve les différents composants de la topologie (figure 37).

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
Spout-generateur-de-phrases	1	1	20	20	0.000	0	0			

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Fail
bolt-affichage	1	1	0	0	0.001	0.833	120	0.000	0	0
bolt-compteur-de-mots	1	1	140	140	0.014	7.286	140	0.000	0	0
bolt-decoupage-en-mots	1	1	100	100	0.000	0.000	0	0.000	0	0

Figure 37. Détails sur la topologie (suite)

Un option très utile dans le cas d'une topologie complexe est la visualisation possible via le bouton Show Visualization (figure 38).

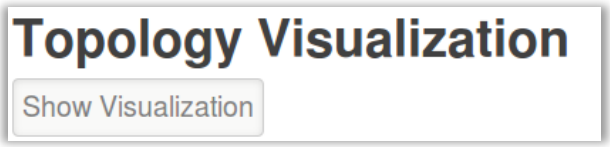

Figure 38. Visualisation de la topologie

On obtient le dessin de la figure 39.

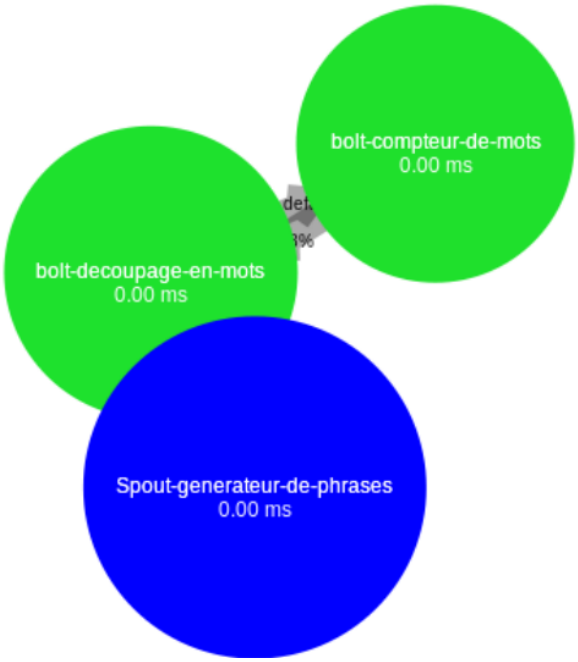

Figure 39. Exemple de visualisation