**Prg4: Worker- Service**

**Projektbeschreibung**

Es gibt eine Vielzahl von Anwendungs- Szenarien, die eine maximale Reaktionszeit für die Abarbeitung einer Aufgabe voraussetzen. Diese festen Antwortzeiten können, bei nicht abschätzbaren Umfang, nicht garantiert werden. Dieses Projekt soll eine Möglichkeit bieten, unter den genannten Bedingungen, feste Antwortzeiten zu garantieren.

Für die Umsetzung der Zielstellung, eine beliebig umfangreiche Aufgabe mit festen Antwortzeiten zu garantieren, sind zwei Komponenten essentiell. Zum einen musste eine allgemeine Repräsentation eines Jobs geschaffen werden. Diese Jobs werden wiederum von einem Worker abgearbeitet. Der Worker- Service arbeitet dabei nach dem Producer- Consumer- Pattern. Dieses wurde unter der Prämisse, dass ein Producer und n Consumer aktiv sind, implementiert.

Weitere Bestandteile bzw. Prämissen des Projektes sind:

* Konfigurierbarkeit (DB- ConnectionStrings, Anzahl maximaler Consumer, Datei- Pfade)
* Datenbank- Anbindung
* Beispielimplementierung.

Nachfolgend wird auf die einzelnen Bestandteile und ihre Konzeption genauer eingegangen.

**Datenbank**

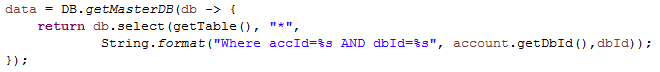
Als Datenbank kommt eine MySQL- Datenbank (mit InnoDb als DB- Engine) zum Einsatz. Auf diese wird mittels der Klasse DB zugriffen. Die Datenbank- Verbindungen werden mittels Connection- Poolings verwaltet. Die DB- Klasse implementiert Basisinteraktionen:

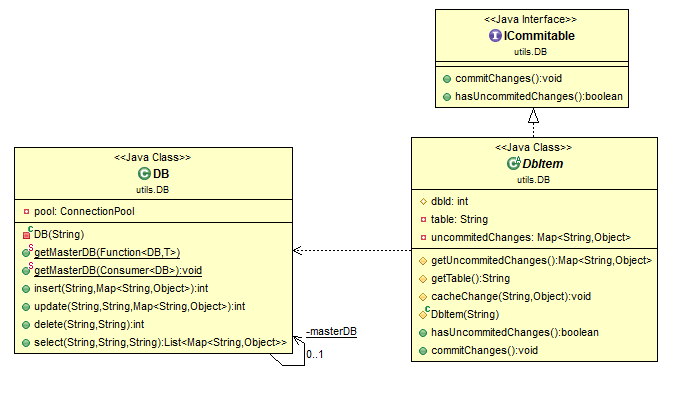
* Select
* Update
* Insert
* Delete.

Es ist mit der vorhandenen Implementierung nicht möglich, eigene SQL- Transaktionen zu schreiben, sondern nur diese vorgefertigten Funktionalitäten zu verwenden. Dies soll im Hinblick auf Schreibaufwand, Wiederverwendbarkeit und Wartbarkeit die Qualität sichern. Somit ist es auch unter vertretbaren Aufwand sehr gut möglich eine andere Datenbanklösung zu verwenden.

Die Konfiguration der Datenbank- Verbindung und des Connection- Pools findet über die zentrale Konfigurationslösung statt (mehr dazu unter Konfiguration).

Beispielhafter DB- Zugriff:





Das Interface ICommitable wird von der Klasse DbItem implementiert und gibt Auskunft über veränderte, nicht in die Datenbank geschriebene Änderungen. Diese Implementierung ist nur für Objekte geeignet, deren Eigenschaften nicht über mehrere Datenbank- Tabellen verteilt sind.

**Job**

Die abstrakte Klasse Job repräsentiert eine abzuarbeitende Aufgabe. Sie implementiert das Runnable- Interface, um die Ausführung in einem separaten Thread zu ermöglichen. Ein Job hat vier Zustände, die durch das Enum State definiert werden.

Ein Job, der auf die Abarbeitung wartet, kann mit der statischen Methode Job#loadNextPending() geladen werden. Es gilt generell, dass beim nicht Auffinden von angefragten Daten null zurückgegeben wird. Dies trifft auch in diesem Fall zu.

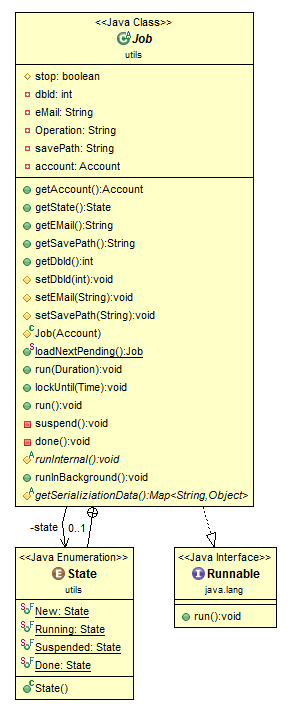
Es gibt drei verschiedene Möglichkeiten einen Job auszuführen:

* Job#run(): Ohne zeitliche Begrenzung; keine Serialisierung in die DB
* Job#run(Duration): Mit zeitlicher Begrenzung; bei Überschreiten der angegeben Zeit wird die Ausführung abgebrochen und der Job in die DB serialisiert.
* Job#runInBackground(): Keine Ausführung; direkte Serialisierung in die DB.

Um einen Job zu implementieren müssen lediglich die abstrakten Methoden runInternal() (hier findet die eigentliche Arbeit des Jobs statt) und getSerializiationData() implementiert werden. Weiterhin muss ein Konstrukter, für die Deserialisierung, mit der Signatur:

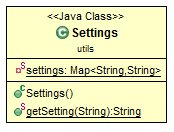
JobImplementation(int dbId, Account rem, String mail, Map<String, Object> args)

vorhanden sein.



**Settings**

Diese Klasse macht die Konfiguration global verfügbar. Die Eigenschaften werden aus der Datei: settings.config gelesen. Es liegt eine Beispiel- Konfigurations- Datei bei, die jedoch noch umbenannt und den eigenen Anforderungen nach angepasst werden muss.

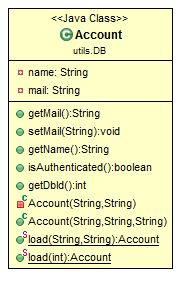


Eigenschaften der Konfiguration können mittels Settings#getSettings(String) abgerufen werden. Änderungen stehen sofort nach Neustart der Applikation zur Verfügung.

**Account**

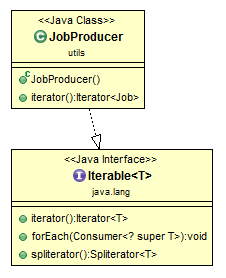
Ein Account dient zur Authentifizierung und zur Zuordnung von Daten, zu einem bestimmten Benutzer.

Die Klasse erweitert die Klasse DbItem und steht somit erst nach einem erfolgreichen Account#commitChanges() zur Verfügung, bzw. gilt erst dann als authentifiziert.



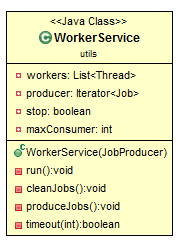
**JobProducer**

Der JobProducer ist dafür zuständig, dem WorkerService bzw. den Consumern Jobs zur Verfügung zu stellen, die abgearbeitet werden soll. Dafür braucht man die Information, ob es noch weitere Jobs gibt und die Fähigkeit einen Job zu laden. Das Iterable- Interface besitzt diese Fähigkeiten, weshalb der JobProducer das Iterable<Job> Interface implementiert. In der momentanen Implementierung gibt es nur einen laufenden Producer, doch es wurde darauf geachtet, dass die Funktionalitäten Thread- sicher implementiert sind. Es sollte kein Problem darstellen, mehrere Producer laufen zu lassen.

****

**WorkerService**

Die maximale Anzahl von parallel laufenden Consumern kann in der settings.config definiert werden. Dabei werden so lange neue Jobs geladen, bis es entweder keine Jobs mehr gibt oder die maximale Anzahl erreicht ist. Man kann den Dienst durch eine beliebige Eingabe stoppen.



Der WorkerService geht wie folgt vor:

* lösche alle beendeten Jobs
* hole neue Jobs bis keine mehr vorhanden sind oder maximale Anzahl erreicht wurde
* Prüfung ob eine Eingabe (Stoppsignal) gesendet wurde

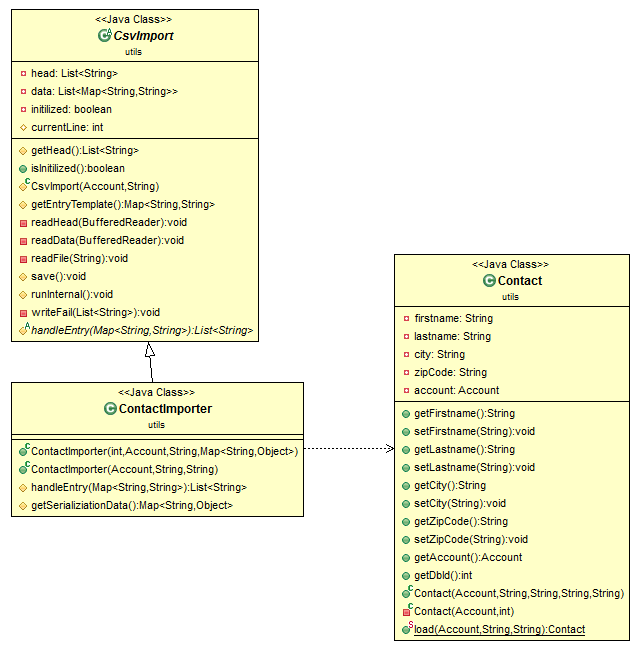
Dieser Vorgang wiederholt sich solange, bis ein Stoppsignal gesendet wurde. Vor jeder Wiederholung werden fünf Sekunden gewartet.

**Beispielimplementierung**

Die Beispielimplementierung umfasst einen Importvorgang von CSV- Dateien. Diese beinhalten Kontaktdaten.

Es wurde eine Basis- Import- Klasse geschaffen. Diese folgt dem Paradigma des zeilenweisen Abarbeitens. Sie erbt von der Job- Klasse und kapselt die benötigten Funktionalitäten zum parsen von CSV- Dateien. Weiterhin definiert sie die Funktion handleEntry(Map<String,String). Diese Funktion wird durch den speziellen Importer (ContactImport) implementiert. Dabei wird zeilenweise der Inhalt der CSV- Datei an diese Funktion gereicht. Somit muss die Implementierung die Verarbeitung und somit, dass Anlegen der neuen Daten gewährleisten. Zusätzlich muss die Serialisierungs- Funktion, sowie der Deserialisierungs- Konstruktor implementiert werden.

Die Klasse Contact ist nur eine einfache Objektrepräsentation eines Kontaktes. Sie implementiert auch die Klasse DbItem.



**Inbetriebnahme**

* alle Referenzen sind in der Datei: pom.xml definiert (Maven Projekt)
* lauffähige MySql- Datenbank
  + entwickelt mit Version 5.7.11 und InnoDB als DB- Engine
  + Zugangsdaten für DB
  + Das DB- Schema liegt als Sql- Script im Ordner src/DBScript bei.
* Die Klasse Application beinhaltet ein Ablauf- Szenario (einzige static main() Methode), dieses kann nach Konfiguration der Datenbank ausgeführt werden.
  + Die beinhaltenden Dateien mock[1..4].csv fassen unterschiedlich viele Kontakte.
  + Es werden 4 Jobs erstellt und mit einer Laufzeit von 10 Sekunden ausgeführt (nacheinander).
  + Aufgrund der Menge von Einträgen sollten 2 Jobs (mindestens 1) länger als 10 Sekunden benötigen und werden in die DB serialisiert.
  + Es startet der WorkerService und sucht nach Jobs
    - Es werden die Jobs deserialisiert und abgearbeitet.
    - Nach Fertigstellung wird der Datenbank- Eintrag des Jobs aktualisiert.
    - Der Worker räumt die fertigen Jobs aus dem Pool.
    - Der Worker sucht weiterhin nach Jobs bis eine Eingabe getätigt wurde.
* Bei Fragen erreichen Sie mich per Email: chr.chart@gmail.com