In dit artikel beschrijft Walter van Iterson, aan de hand van een voorbeeldapplicatie, drie verschillende manieren om data op te slaan. Naast XML behandelt hij objectrelationele mapping. Dit wordt veel gebruikt in J2EE-omgevingen, maar is in een lichte variant ook toepasbaar voor desktopapplicaties. Verder komt prevalence aan de orde, een uitbreiding van de normale serialisatie. Dit artikel geeft inzicht in de alternatieven, zodat gebruikers in de toekomst weten wat in bepaalde situaties de beste oplossing is.

thema

Drie alternatieven voor persistentie

De beste manier om data op te slaan

In vrijwel alle programma's wil de gebruiker data op de een of andere manier opslaan. In een serveromgeving kan de applicatieserver hiervoor zorgen. Bij desktopapplicaties is zo'n applicatieserver niet aanwezig. Een voor de hand liggende manier om de data dan toch op te slaan is XML-bestanden te schrijven, maar dit is zeker niet de enige manier. XML is in veel situaties zeker een goed alternatief, maar biedt niet in alle gevallen de optimale oplossing. Er zijn meer mogelijkheden, elk met hun eigen sterke en zwakke kanten.

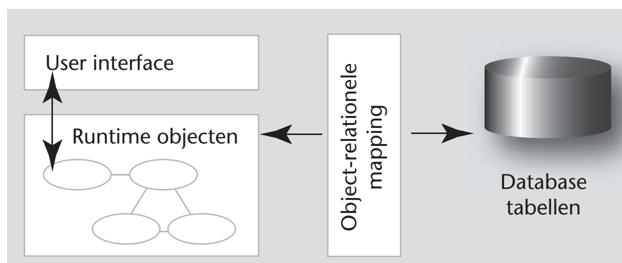
Als voorbeeld hanteren we hier een typische desktopapplicatie: een programma waarmee een muziekverzameling kan worden beheerd. Een paar eigenschappen van dit programma, die bij het ontwerpen van het datamodel van invloed zijn:

- De hoeveelheid te beheren data valt mee, de data past goed in het geheugen.
- Er zijn relatief weinig afhankelijkheden van externe systemen. Misschien wordt cddb.com een keer geraadpleegd om de details van een nummer op te vragen, maar daar blijft het wel bij.
- Er is maar één gebruiker tegelijkertijd, dus de dynamiek van de data is vrij beperkt.

OBJECTRELATIONELE MAPPING Objectrelationele mapping (ORM) wordt veel gebruikt in serversystemen. Aan de achterkant van het systeem worden de data opgeslagen in een database. De ORM-tool maakt de vertaling tussen runtime-objecten en records in de database. Als je een runtime-object verandert, leidt dit automatisch tot een aanpassing van één of meerdere tabellen in de database. Schematisch ziet dit er als volgt uit:

PRODUCTEN ORM-tools laten de gebruiker vaak vrij in de keuze van database. Bij kleine programma's heeft een lichtgewicht database de voorkeur. Zo kan bijvoorbeeld hsqldb (www.hsqldb.org) gebruikt worden, de database die ook deel uitmaakt van OpenOffice 2.0. In de wereld van de ORM-tools zijn er op dit moment twee grote stromingen: Hibernate (www.hibernate.org), een open source-project dat door zijn populariteit inmiddels een *de facto*-standaard lijkt, en JDO (www.jdocentral.com), een standaardisatie-initiatief waar veel leveranciers aan deelnemen. In de volgende versie van JDO, JDO2, zullen deze twee stromingen deels samenkomen.

TOEPASSING Vanuit hun achtergrond in de serversystemen zijn de tools op veel verschillende manieren te gebruiken. In de praktijk komt dit neer op heel veel werk om de boel goed te configureren. ORM-tools hebben typisch twee soorten configuratiefiles: één voor de mapping, de vertaling tussen runtime-objecten en databasetabellen, en één voor de configuratie van de ORM-tool zelf: wat voor soort database hangt er onder, hoe is die te bereiken, enzovoort.



FIGUUR 1. De ORM-tool maakt de vertaling tussen runtime-objecten en records in de database.

Een configuratiefile van Hibernate, met een hsqldb-database ziet er als volgt uit:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-
//Hibernate/Hibernate Configuration DTD 3.0//
EN"
     "http://hibernate.sourceforge.net/hiber-
nate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!—Instellingen van de database -->
    <property name="connection.driver_
class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:
hsqldb:data/songdb</property>
    <property name="connection.username">sa</
property>
    <property name="connection.password"></
property>

    <!— Iedere database heeft net een andere
interpretatie van SQL.
        Dit is op te vangen door een SQL
dialect in te stellen -->
    <property name="dialect">org.hibernate.
dialect.HSQLDialect</property>

    <!-- Mapping files voor de verschillende
classes.
        Conventie is 1 mapping file per
relevante class -->
    <mapping resource="Song.hbm.xml"/>
    <mapping resource="Artist.hbm.xml"/>
    <mapping resource="Album.hbm.xml"/>
    ...
  </session-factory>
</hibernate-configuration>
```

Een mapping file voor een nummer (Song.hbm.xml) ziet er zo uit:

```
<hibernate-mapping>
  <class name="Song">
       <!-- persisteer instanties van class
Song -->
    <id name="id" column="songId">
     <!-- met een unieke id
        <generator class="increment"/>
     <!-- automatisch gegenereerd -->
    </id>
    <property name="title"/>
      <!-- sla het veld "title" op -->

    <array name="artists" table="SongArtist">
<!-- persisteer de artiesten

            in een tabel "SongArtist" -->
      <key column="songId"/>
      <!-- gebruik de songId als key -->
      <list-index column="sortOrder"/>
     <!-- op volgorde -->
      <many-to-many column="artistId"
      <!-- aan de andere kant van de relatie
zit
                   class="Artist"/>
         <!-- een Artist, met als key de
artistId -->
    </array>
  </class>
</hibernate-mapping>
```

Voor- en nadelen

+ Goed schaalbaar als het systeem groter wordt.
+ Ook snel bij opstarten met grote hoeveelheden data, omdat de data pas geladen wordt als deze nodig is.
+ Grote community, veel voorbeelden en support (wel veel focus op serversystemen).
- Snel overkill voor kleine applicaties.
- Bij veel acties op grotere hoeveelheden data (dus niet bij de muziekverzameling) gaat de mapping op de onderliggende databasestructuur ten koste van de performance van de applicatie.

Object-relationele mapping is een zeer krachtig concept, maar bij een beperkte hoeveelheid vrij statische data is het een flinke overkill. Een alternatief, aan de andere kant van het spectrum, is de data zelf naar een file schrijven.

Eigen formaat Over data opslaan in een eigen formaat is veel te zeggen. Enerzijds biedt een eigen formaat de meeste mogelijkheden, anderzijds moet alles wel door de gebruiker zelf geïmplementeerd worden, omdat in principe van nul af aan wordt begonnen. Het staat de gebruiker bij een eigen formaat natuurlijk volledig vrij om te kiezen hoe hij de data wil schrijven. Hier wordt een XML-bestand als voorbeeld genomen.

Lezen XML-bestanden kunnen op verschillende manieren worden gelezen (parsen). De twee bekendste soorten XML-parsers zijn de DOM-parser en de SAX-parser. De DOM-parser bouwt een boomstructuur met de inhoud van het XML-document, die vervolgens kan worden doorlopen, waarna de datastructuur gevuld kan worden. De SAX-parser werkt op basis van een callback-mechanisme: zodra de parser een XML-element tegenkomt wordt de methode ContentHandler.startElement() aangeroepen, die de gebruiker kan toepassen om een klein deel van de datastructuur te vullen.

Wanneer een DOM of SAX-parser wordt gebruikt moet wel de vertaling van XML-elementen naar de eigen datastructuur gemaakt worden. Het Apache Digester-project (jakarta.apache.org/commons/digester) breidt de SAX- parser uit met de mogelijkheid om methoden aan te roepen op basis van XML-elementen die in het bestand staan. Veronderstel, dat een deel van het XML-bestand beschrijft welke nummers er zijn:

```
<songs>
  <song id=”16” title=”Vertigo”/>
  <song id=”29” title=”American Life”/>
</songs>
```

Met de volgende configuratie zet Digester dit om in een runtime datastructuur:

```
…
digester.addObjectCreate("songs", SongList.
class);  // maak bij element “songs” een
SongList
digester.addObjectCreate("songs/song", Song.
class);  // maak bij element “song” een Song
digester.addSetNext("songs/song", "addSong");
 // roep aSongList.addSong aan na een song
digester.addCallMethod("songs/song", "setId",
1);   // roep aSong.setId aan met 1 parame-
ter:
digester.addCallParam("songs/song", 0, “id”);
 // de waarde van het attribute “id”
digester.addCallMethod("songs/song", "setTit-
le", 1); // enz.
digester.addCallParam("songs/song", 0,
“title”);
…
```

Schrijven Een mogelijkheid is een XML-bestand schrijven door in de code een DOM-boomstructuur te bouwen, en deze weg te schrijven met een Transformer. Een andere mogelijkheid is de XML rechtstreeks schrijven is de volgende:

```
public void writeSong(Song aSong,
BufferedWriter writer) throws IOException {
    writer.write(“<song id=\"”);
    writer.write(aSong.getId());
    writer.write(“\" title=\"”);
    writer.write(aSong.getTitle());
    writer.write(“\">\n”);
    for (Artist anArtist:aSong.getArtists())
{
        writer.write(“\t<artist id=\"”);
        writer.write(anArtist.getId());
        writer.write(“\"/>\n”);
    }
    writer.write(“</song>”);
}
```

Toepassing Het mooie van een eigen formaat is, dat de gebruiker het volledig kan optimaliseren voor de toepassing waar het voor gebruikt gaat worden. Toch is het handig om bij het implementeren een paar regels voor de mapping te hanteren. Zo kan er bijvoorbeeld worden uitgegaan van de volgende richtlijnen:

- Per relevant Java-object een XML-element: `<song>…</song>`
- Implementeer velden die één keer in een Java-object voorkomen als XML- attributen, met als attribuutnaam de naam van het veld: `<artist name=”Dido”>`
- Sla referenties die vaker voorkomen binnen één object op als geneste XML- elementen:

```
<album>
  <beschrijving>Live opname van het …</
beschrijving>
  <beschrijving>Gekregen voor m’n 31ste…</
beschrijving>
</album>)
```

- Implementeer referenties naar andere objecten door elke class een id te geven, en die te gebruiken voor verwijzingen:

```
<song album=”12”>
…
<album id=”12”>
```

- Met een HashMap van id naar het Album kan hier het probleem omzeild worden dat het Album-object nog niet bestaat als het Song-object gemaakt wordt.

Voor- en nadelen

\+ Met een eigen dataformaat heeft de gebruiker volledige controle over wat er in de bestanden komt te staan.
\+ Gebruikers zijn beter bekend met databestanden dan met databases. Ze kunnen de data files e-mailen, backups maken, enzovoort.
\+ XML-bestanden zijn met een XSLT-processor te converteren. Zo kan de data bijvoorbeeld omgezet worden naar een tabelvorm en op een website geplaatst worden.
\+ Het is vaak leuk om het wiel zelf uit te vinden.
\- Alles zelf doen betekent ook zelf de fouten oplossen.
\- De gebruiker is niet direct bezig met waar het programma eigenlijk om draait.
\- Het XML-formaat heeft van nature vrij veel overhead. Bij grotere hoeveelheden data gaat dit ten koste van de performance.

In plaats van een eigen dataformaat te definiëren kan de gebruiker natuurlijk ook gebruikmaken van serialisatie om de data op disk te zetten. Het grote voordeel daarvan is dat niet hoeft worden opgegeven welk veld op wat voor manier weggeschreven moet worden. Ook met serialisatie wordt data nog steeds alles-of-niets weggeschreven. Een techniek die prevalence heet kan worden gebruikt wanneer alle veranderingen wel meteen opgeslagen moeten worden, maar niet alle data na elke verandering weggeschreven moet worden.

Prevalence Bij prevalence wordt eens in de zoveel tijd een snapshot gemaakt. De volledige datastructuur wordt dan geserialiseerd. Naast deze snapshot worden de veranderingen opgeslagen zodra ze plaatsvinden. Een verandering is dus ook een object dat geserialiseerd kan worden.

Hier volgt een voorbeeld van hoe je een verandering als een transactie-object kunt modelleren:

Waar normaal gesproken een nummer uit de muziekverzameling gezet zou worden met:

```
myMusicCollection.addSong(mySong)
```

Implementeer de Transactie-interface, met in executeOn() de eigenlijke verandering:

```
import org.prevayler.Transaction;

public class AddSongTransaction implements
Transaction {
    private Song songToAdd;

    public AddSongTransaction(Song songToAdd)
{
        this.songToAdd = songToAdd;
    }

    public void executeOn(Object prevalentSy-
stem, Date ignored) {
        ((MusicCollection)prevalentSystem).
addSong(songToAdd);
    }
}
```

Vervolgens zorgt de execute() method van de prevalence engine ervoor dat deze transactie uitgevoerd wordt:

```
import org.prevayler.Prevayler;
import org.prevayler.PrevaylerFactory;

    ...
    // initialiseer prevalence engine, dit
laadt de eerder opgeslagen data
    // in directory myMusic
    Prevayler prevayler = PrevaylerFactory.cr
eatePrevayler(myMusicCollection, "myMusic");
    ...
    // en laat de engine het nummer aan de
muziekverzameling toevoegen
    prevayler.execute(new AddSongTransaction(
mySong));
```

Als het systeem (al dan niet vrijwillig) gestopt wordt, staan er twee files op de harde schijf: één met de snapshot van een paar uur geleden, en één met alle veranderingen na die tijd. Bij het opstarten van het systeem wordt eerst de laatste snapshot hersteld, en vervolgens alle veranderingen die na die snapshot plaats hebben gevonden. Nadat deze veranderingen zijn uitgevoerd, is de toestand weer hetzelfde als voordat het systeem gestopt was.

PRODUCTEN Eenmaal bekend met dit concept is het vrij eenvoudig zelf te implementeren, om optimaal aan te sluiten op de rest van de applicatie. Voor zover bekend, is Prevayler (www.prevayler.org) de enige Java-library die dit al voor de gebruiker heeft geïmplementeerd.

TOEPASSING Omdat alle data in het geheugen staat, nemen transacties (de veranderingen) niet veel tijd in beslag. Afhankelijk van de complexiteit van de datastructuur kun je denken in de richting van 4.000 transacties per seconde op een 1GHz desktop systeem. Door deze snelheid kan de prevalence-engine transacties single-threaded uitvoeren, wat een hoop threading-problemen bij voorbaat uitsluit.

VOOR- EN NADELEN

+ Eenvoudige configuratie, De mapping van de data structuur op het filesysteem wordt door de Java-runtime-omgeving gedaan.
+ Data wordt ‘continu’ opgeslagen, dus na een crash heeft het systeem dezelfde data als vlak ervoor.
+ Er is weinig code nodig om dit concept te implementeren, wat de kans op fouten beperkt. De jar file van Prevayler is 47 kb.
+ Door veranderingen als transacties te modelleren is undo-functionaliteit vrij eenvoudig te implementeren.

- Het grootste nadeel van deze techniek voor desktopapplicaties is dat de gebruiker wordt gedwongen om veranderingen in de data als transacties te modelleren.
- Serialisatie leidt nogal eens tot problemen met verschillende versies van classes.
- Bij vele megabytes aan data en veel veranderingen (grote dynamische systemen, dus niet de muziekverzameling) kan het opstarten enige tijd duren, voordat alle data in het geheugen is geladen.

TEVREDENHEID EINDGEBRUIKERS De drie beschreven alternatieven voor persistentie zijn conceptueel totaal verschillend. Toch leiden ze uiteindelijk alledrie tot hetzelfde resultaat: de data die in het geheugen staat wordt op de harde schijf gezet, en kan later weer in het geheugen gelezen worden. De drie behandelde alternatieven, en de vele die hier niet aan de orde gekomen zijn, hebben door de verschillende aanpak heel andere sterke en zwakke kanten. De keuze voor een andere persistentietechniek kan flinke gevolgen hebben op de manier waarop het datamodel van een applicatie opgebouwd is. Uiteindelijk beïnvloedt deze keuze ook de installatie, het gebruiksgemak en de performance, en dus de tevredenheid van de eindgebruikers.

Walter van Iterson is consultant ICT bij Ordina.