Hypersistence Optimizer – czy optymalizacja warstwy dostępu do danych może być prosta?

Za większością aplikacji oraz serwisów internetowych stoi źródło danych (np. baza danych lub pliki z danymi), które przechowuje wszelkiego rodzaju informacje potrzebne do poprawnego funkcjonowania systemu. Zatem komunikacja aplikacji ze źródłem danych jest kluczowym elementem systemu. Za tę komunikację odpowiada warstwa dostępu do danych. W javowym świecie popularnym sposobem implementacji tej warstwy jest mapowanie obiektowo-relacyjne, które polega na odzwierciedleniu obiektowej architektury systemu na bazę danych o relacyjnym charakterze. Najpopularniejszym frameworkiem wykorzystywanym przy mapowaniu obiektowo-relacyjnym jest Hibernate, który jest implementacją standardu JPA. Hibernate jest rozbudowanym i potężnym narzędziem, a co za tym idzie, nie jest prosty w użyciu. Podczas konfiguracji oraz implementacji można popełnić wiele błędów, które w znacznym stopniu mogą wpłynąć negatywnie na wydajność budowanego systemu. Szukanie błędów oraz późniejsza optymalizacja może być bardzo czasochłonna. Z pomocą przychodzi Hypersistence Optimizer!

*„Imagine having a tool that can **automatically** detect if you are using JPA and Hibernate properly.”* – Hypersistence Optimizer

Hypersistence Optimizer – co to takiego?

Hypersistence Optimizer jest narzędziem do automatycznej analizy warstwy dostępu do danych, które wspiera szeroki zakres wersji Hibernate’a (v.3.3 – 5.4). Procesowi analizy podlega konfiguracja Hibernate’a, mapowanie encji, zapytania do bazy danych oraz Persistance Context. Na

podstawie przeprowadzonej analizy otrzymujemy szczegółowy raport na temat problemów znalezionych w aplikacji. Taki raport zawiera między innymi informacje o istotności znalezionego problemu, jego lokalizacji, krótki opis problemu oraz odnośnik do bardziej szczegółowego objaśnienia wraz ze wskazówkami i przykładami. Narzędzie zostało stworzone przez firmę Hypersistence i jest ciągle rozwijane. Aktualnie narzędzie jest w stanie wykryć ponad 50 różnego rodzaju błędów. Hypersistence Oprimizer występuje w dwóch wersjach: pełna i trial. Trial jest darmowy, ale za to bardzo okrojony w porównaniu z pełną wersją. Wersja próbna wyłapuje tylko rodzaj i ilość błędów, nie wskazując gdzie te błędy występują i jak sobie z nimi poradzić. Przed zakupem pełnej wersji dobrym pomysłem będzie sprawdzenie za pomocą wersji testowej czy w tworzonej aplikacji występują błędy.

Pierwsze kroki

Proces instalacji składa się z kilku prostych kroków. Przed instalacją należy upewnić się czy na komputerze zainstalowany jest Maven, który jest potrzebny do zainstalowania narzędzia. Po pobraniu i rozpakowaniu archiwum wystarczy uruchomić plik wsadowy „maven-install", który znajduje się w wypakowanym folderze. Hypersistence Optimizer zostanie dodany do repozytorium Mavena. Następnym krokiem jest dodanie zależności do pliku pom.xml w projekcie:

```
1.  <dependency>
2.      <groupId>io.hypersistence</groupId>
3.      <artifactId>hypersistence-optimizer</artifactId>
4.      <version>2.1.1</version>
5.  </dependency>
```

Po tym kroku, narzędzie jest już gotowe do użytku. Włączanie analizy odbywa się z poziomu testów jednostkowych. Należy stworzyć test oraz umieścić w nim inicjalizację Optimizera. Może to wyglądać tak:

```
1.   class HypersistenceTest {
2.
3.       @PersistenceUnit
4.       private EntityManagerFactory entityManagerFactory;
5.
6.       @PostConstruct
7.       public void init() {
8.           new HypersistenceOptimizer(new JpaConfig(entityManagerFactory));
9.       }
10.
11.      @Test
12.      void someTest() {
13.      }
14.  }
```

Jak widzimy użycie Hypersistence Optimizera jest bardzo proste. Podczas inicjalizacji należy podać tylko EntityManager, z którego narzędzie pobiera potrzebne do analiza metadane. Jest to oczywiście

najbardziej podstawowa konfiguracja. Możliwości konfiguracji są bardziej rozbudowane. Można między innymi określić jaki obszar warstwy dostępu do danych ma zostać przeanalizowany (konfiguracja, mapowanie lub zapytania), jakie eventy mają zostać wyłapane lub określić co ma się stać z eventem, który został wykryty. Każdy napotkany problem podczas analizy wywołuje zdarzenie (event) związane z właśnie sprawdzaną regułą. Zatem wszystkie problemy, które zostaną wykryte zwracane są w postaci zdarzeń (eventów), które można przypisać do konkretnej reguły i opisu. Domyślnie narzędzie analizuje wszystkie obszary oraz wyłapuje wszystkie napotkane eventy. Autorzy dostarczają szczegółowy opis konfiguracji, za pomocą którego można przystosować narzędzie do swoich potrzeb.

Ostatnim krokiem jest opakowanie domyślnej konfiguracji Hibernate SessionFactory dostarczonym przez twórców dekoratorem. Ten krok jest konieczny jeżeli chcemy aby zapytania były analizowane w czasie rzeczywistym. Gdy korzystamy z Hibernate 5 wystarczy skopiować plik, który znajduje się w folderze ..\configs\META-INF\services do folderu ..\src\test\resources bieżącego projektu. Tak skonfigurowane narzędzie będzie działać w środowisku testowym nie powodując żadnych problemów w środowisku produkcyjnym. W przypadku starszych wersji Hibernate konieczne jest manualne opakowanie SessionFactory. Szczegółowy opis znajduje się w dostarczonym przewodniku konfiguracyjnym.

Optimizer w akcji

Po uruchomieniu testu jednostkowego, w którym został skonfigurowany Hypersistence Optimizer, w konsoli pojawia się raport. Narzędzie testowane było na małej aplikacji do rezerwacji samochodów firmowych. Otrzymany raport wygląda następująco:

```
CRITICAL - IdentityGeneratorEvent - The [id] identifier attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the [IdentityGenerator] strategy, which prevents
Hibernate from enabling JDBC batch inserts. Consider using the SEQUENCE identifier strategy
instead. For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#IdentityGeneratorEvent

MAJOR - EnumTypeStringEvent - The [fuelType] enum attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the EnumType.STRING strategy, which has a bigger
memory footprint than EnumType.ORDINAL. For more info about this event, check out this User
Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#EnumTypeStringEvent

CRITICAL - BidirectionalSynchronizationEvent - The [reservations] bidirectional association in the
[pl.fis.carrs.domain.model.Vehicle] entity requires both ends to be synchronized. Consider adding
the [addReservation(pl.fis.carrs.domain.model.Reservation reservation)] and
[removeReservation(pl.fis.carrs.domain.model.Reservation reservation)] synchronization methods.
For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#BidirectionalSynchronizationEvent
```

```
CRITICAL - EagerFetchingEvent - The [vehicle] attribute in the
[pl.fis.carrs.domain.model.Reservation] entity uses eager fetching. Consider using a lazy fetching
which, not only that is more efficient, but it is way more flexible when it comes to fetching data.
For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#EagerFetchingEvent

MAJOR - SkipAutoCommitCheckEvent - You should set the
[hibernate.connection.provider_disables_autocommit] configuration property to [true] while also
making sure that the underlying DataSource is configured to disable the auto-commit flag
whenever a new Connection is being acquired. For more info about this event, check out this User
Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#SkipAutoCommitCheckEvent

CRITICAL - JdbcBatchSizeEvent - If you set the [hibernate.jdbc.batch_size] configuration property
to a value greater than 1 (usually between 5 and 30), Hibernate can then execute SQL statements
in batches, therefore reducing the number of database network roundtrips. For more info about
this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-
optimizer/docs/user-guide/#JdbcBatchSizeEvent

CRITICAL - QueryPaginationCollectionFetchingEvent - You should set the
[hibernate.query.fail_on_pagination_over_collection_fetch] configuration property to the value of
[true], as Hibernate can then prevent in-memory pagination when join fetching a child entity
collection. For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#QueryPaginationCollectionFetchingEvent

MAJOR - QueryInClauseParameterPaddingEvent - You should set the
[hibernate.query.in_clause_parameter_padding] configuration property to the value of [true], as
Hibernate entity queries can then make better use of statement caching and fewer entity queries
will have to be compiled while varying the number of parameters passed to the in query clause.
For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#QueryInClauseParameterPaddingEvent

8 issues were found: 0 BLOCKER, 5 CRITICAL, 3 MAJOR, 0 MINOR
```

Na podstawie informacji zawartych w raporcie możemy zacząć optymalizować warstwę dostępu do danych. Rozpatrzmy na przykład błąd EagerFetchingEvent, który został zakwalifikowany jako błąd krytyczny. Relacja między rezerwacją a samochodem została zdefiniowana jako wiele-do-jednego jednak nie została podana strategia ładowania kolekcji vehicle.

```
1.  @ManyToOne
2.  private Vehicle vehicle;
```

W przypadku używania relacji wiele-do-jednego oraz jeden-do-jednego domyślną strategią ładowania kolekcji jest FetchType.EAGER. Ładowanie kolekcji EAGER oznacza, że dane pobierane są w całości

w momencie pobierania ich rodzica, nawet jeśli ich w ogóle nie potrzebujemy. Zalecane jest w takim przypadku używać strategii FetchType.LAZY, która pobiera kolekcję w momencie gdy jest ona potrzebna. Kod po optymalizacji mógłbym wyglądać następująco:

```
1. @ManyToOne (fetch = FetchType.LAZY)
2. @JoinColumn (name = "vehicle_id")
3. private Vehicle vehicle;
```

Hypersistence Optimizer wyłapuje potencjalne zagrożenia w warstwie dostępu do danych, które mogą (ale nie muszą!) powodować problemy wydajnościowe. Jednakże są to tylko wskazówki, które można uwzględnić aby wyeliminować istniejące lub potencjalne problemy. Konkretne rozwiązania zależą od złożoności budowanego systemu, dlatego każdy wyłapany problem należy rozważać indywidualnie. Przykładem może tu być EnumTypeStringEvent. W tym przypadku zalecane jest aby typy wyliczeniowe enum zapisywać w bazie w postaci EnumType.ORDINAL. W testowanej aplikacji enumy zapisywane są jako EnumType.STRING. Przejście z EnumType.STRING na EnumType.ORDINAL pozwoli zaoszczędzić trochę pamięci, jednak czytelność kodu oraz czytelność informacji w bazie danych spada. Tu wybór należy do programisty czy zachować czystość w kodzie albo zaoszczędzić trochę pamięci.

Druga część raportu zawiera wyłapane problemy związane z zapytaniami (pojawia się tylko po uprzednim skonfigurowaniu Hibernate SessionFactory):

```
CRITICAL - PaginationWithoutOrderByEvent - The [select v from VEHICLES v ] query uses pagination without an ORDER BY clause. Therefore, the result is not deterministic since SQL does not guarantee any particular ordering unless an ORDER BY clause is being used. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PaginationWithoutOrderByEvent

CRITICAL - PassDistinctThroughEvent - The [SELECT DISTINCT v FROM VEHICLES v left join v.reservations res where not exists… ] query uses DISTINCT to deduplicate the returned entities. However, without setting the [hibernate.query.passDistinctThrough] JPA query hint to [false], the underlying SQL statement will also contain DISTINCT, which will incur extra sorting and duplication removal execution stages. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PassDistinctThroughEvent
```

W aktualnej wersji Optimizera wyłapywane są na razie tylko 2 rodzaje eventów, jeżeli chodzi o analizę zapytań: PaginationWithoutOrderByEvent oraz PassDistinctThroughEvent. Na przykładzie widzimy, że nawet w tak prostym zapytaniu „select v from VEHICLES v” można popełnić błąd. W przypadku ograniczenia wyników za pomocą setMaxResults należy dodać do zapytania klauzulę ORDER BY.

Podsumowanie

Ogromna wiedza oraz zaangażowanie autora – Vlada Mihalcea jako rzecznika programistów dla projektu Hibernate, zaowocowało powstaniem świetnego narzędzia do automatycznej analizy warstwy dostępu do danych, jakim jest Hypersistence Optimizer. Pozwala ono zaoszczędzić programistom mnóstwo czasu i siwych włosów podczas szukania przyczyn słabej wydajności aplikacji. Ponadto szczegółowe opisy problemów wraz z przykładami mogą służyć jako baza wiedzy dla każdego programisty.

Autor: Denis Wiesner

Inżynier, programista java z kilkuletnim doświadczeniem zawodowym w backendzie oraz frontendzie. Zainteresowania: metody numeryczne, programowanie równoległe, nowe technologie.

#java #hibernate #jpa #orm #hypersistenceOptimizer #optymalizacja