Engineer

ORM

et bibliotek, der kobler tabeller til objekter i koden

Hvad det er

En ORM, kort for object-relational mapper, er et bibliotek, der kobler tabellerne i en relationel database til klasser i applikationskoden og rækkerne i dem til objekter. Koden beder om en forfatter og får et `Author` -objekt tilbage, ændrer en af dets attributter og gemmer det, og ORM-biblioteket skriver de `SELECT` -, `INSERT` - og `UPDATE` -sætninger, der gør arbejdet. Djangos modeller og SQLAlchemys ORM, begge skrevet i Python, er to kendte eksempler, og det er dem, denne artikel bygger på.

Kortlægningen er nødvendig, fordi de to sider organiserer data forskelligt. Martin Fowler siger det ligeud i sin beskrivelse af mønstret Data Mapper: objekter og relationelle databaser strukturerer data på hver sin måde, og dele af en objektmodel, som samlinger og nedarvning, har intet modstykke i en relationel database. At flytte data mellem de to bliver et problem i sig selv, og en ORM er et færdigt svar på det.

Delene af det svar fik navne i Fowlers bog Patterns of Enterprise Application Architecture fra 2003, hvis mønsterkatalog ligger frit tilgængeligt på nettet. To mønstre beskriver den overordnede form. Active Record er et objekt, der indkapsler én række i en tabel, selv står for sin databaseadgang og bærer domænelogikken for de data; ifølge Djangos designfilosofi følger modellerne i Django netop det mønster. Data Mapper er et separat lag, der flytter data mellem objekterne og databasen og holder dem uafhængige af hinanden, så objekterne ikke behøver at vide, at der overhovedet findes en database. Rundt om dem ligger en række mindre mønstre. Identity Map sørger for, at hvert objekt kun indlæses én gang. Unit of Work holder en liste over de objekter, et stykke arbejde har ændret, og skriver ændringerne ud samlet. Lazy Load er et objekt, der kun rummer en del af de data, koden skal bruge, men ved, hvordan resten hentes. SQLAlchemys dokumentation nævner to af dem direkte: dens `Session`

holder indlæste objekter i et identity map, med kun ét objekt pr. primærnøgle, og dens måde at samle ændringer og flushe dem til databasen på er unit of work-mønstret.

Brug begrebet i en beslutning

Brug »Hvad er en ORM?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Den del, brugerne aldrig ser: hvor data ligger, hvilke regler der gælder, og hvordan andre systemer må tale med den.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

Den første grund er den mængde kode, en ORM sparer. I sit essay ORM Hate fra 2012 kalder Fowler problemet med objektrelationel mapping svært: det handler om at holde to vidt forskellige repræsentationer af de samme data i takt, og en ændring på den ene side skal føres over til den anden. Hans skøn er, at en ORM klarer omkring 80 til 90 % af mappingproblemerne, og han mener, at et framework, der sparer et team for 80 % af det arbejde, er værd at have, også selv om det stopper dér. Hver tabel, teamet ikke mapper i hånden, er ét sted mindre, hvor en kolonne kan blive læst ind i den forkerte attribut.

Den anden grund er sammenhæng inde i et kørende program. Med et identity map returnerer to databaseforespørgsler, der finder den samme række, det samme objekt, så en ændring gennem den ene reference kan ses gennem den anden; SQLAlchemys dokumentation viser det ved at hente den samme primærnøgle to gange og få præcis det samme objekt tilbage. Med en unit of work samler ORM-biblioteket ventende indsættelser, opdateringer og sletninger og skriver dem ifølge SQLAlchemys funktionsliste ud i én samlet omgang, så applikationskoden ikke selv skal huske, hvilke objekter den har rørt. Django vælger med vilje det modsatte: frameworkets designfilosofi kræver et eksplicit kald til `save()` i stedet for, at frameworket gemmer i det skjulte, så der sendes så få SQL-sætninger som muligt.

Den tredje grund er, hvor grænsen går. Fowler beskriver en god ORM som en, der tager sig af det kedelige arbejde, men giver adgang til et mandehul ned til SQL, når der er brug for det, og han siger ligeud, at den, der bygger på en relationel database, skal vide, hvordan sådan en virker. Djangos designfilosofi

siger det samme fra frameworkets side: databasens API skal vide, at det er en genvej og ikke nødvendigvis hele svaret, og frameworket skal gøre det let at skrive egen SQL, hvad enten det er hele sætninger eller blot en `WHERE` -betingelse. En ORM gør sig fortjent til sin plads på den rutineprægede del af dataadgangen; resten er der, hvor et team bruger sine SQL-evner.

Sådan ser det ud i praksis

Den omkostning, en ORM oftest skjuler, er antallet af forespørgsler, den sender. I SQLAlchemy 2.1 indlæses relationer dovent som standard: de relaterede objekter hentes første gang, koden rører attributten. Dokumentationen siger det ligeud: for N indlæste objekter betyder adgang til deres dovent indlæste attributter N+1 `SELECT` -sætninger. Eksemplet herunder forudsætter, at klasserne `Author` og `Book` allerede er mappet med en en-til-mange-relation, `Author.books` , og gennemløber forfatterne på tre måder.

```
from sqlalchemy import select
from sqlalchemy.orm import raiseload, selectinload

# Standarden: én SELECT til forfatterne og så én mere for hver
# forfatter, hvis bøger løkken rører, i alt N+1 sætninger.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN-indlæsning: en ekstra SELECT henter alle forfatternes
# bøger på én gang med deres primærnøgler i en IN-betingelse.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise-indlæsning: adgang til author.books giver nu en fejl
# i stedet for stille at sende endnu en forespørgsel.
stmt = select(Author).options(raiseload(Author.books))
```

Til en-til-mange- og mange-til-mange-samlinger kalder dokumentationen `selectinload()` generelt den bedste strategi. `joinedload()` , som den kalder den mest alsidige strategi, føjer en `JOIN` til den oprindelige sætning, som standard en `LEFT OUTER JOIN` , så de relaterede rækker kommer med i det samme resultatsæt. `raiseload()` findes for at beskytte mod uønskede indlæsninger, og jokertegnsformen `raiseload("*")` udvider beskyttelsen til de øvrige relationer, også dem på de objekter, sætningen faktisk henter.

Django 6.1 giver det samme valg sit eget navn: fetch modes. Når koden rører et felt, som den oprindelige forespørgsel ikke hentede, for eksempel en fremmednøgle, en en-til-en-relation eller et felt, der er udeladt med `defer()` eller `only()`, henter standardtilstanden `FETCH_ONE` det for netop den ene instans, og dokumentationen kalder selv resultatet for N+1-problemet. `FETCH_PEERS`, der er nyt i 6.1, henter det manglende felt for instansen og for alle de andre instanser fra det samme `QuerySet` ud fra den antagelse, at kode, der har brug for et felt på én række, som regel har brug for det på dem alle. `FETCH_RAISE` udløser i stedet `FieldFetchBlocked`. En tilstand sættes pr. `QuerySet`, som i `Book.objects.fetch_mode(models.FETCH_PEERS)`, eller som standard for en model gennem en tilpasset manager, og Django kopierer den videre til de relaterede objekter, det henter. Hvor `FETCH_PEERS` ikke er passende eller effektiv nok, henviser optimeringsguiden til `select_related()` og `prefetch_related()`.

Virksomhedens egne sider trækker grænsen længere ind mod databasen end de fleste. Dens oversigt over teknologier behandler SQL som et fuldgyldigt sprog og ikke som noget, en ORM genererer, fordi det meste af det performancearbejde, virksomheden hentes ind til, ligger dér. I sit arbejde hos NextMariner førte grundlæggeren tanken helt igennem med et funktionsbaseret datalag i PostgreSQL: hver operation, appen kan udføre, er én af 1.275 lagrede funktioner fordelt på 34 skemaer, og der er slet ingen direkte tabeladgang fra Go. Den rolle, API-laget logger ind som, har `EXECUTE` på de funktioner og `USAGE` på deres skemaer og intet andet, så den kan ikke røre en tabel direkte. Siden lægger ikke skjul på prisen: et nyt felt kræver en ændring i en tabel såvel som i en funktion og ikke blot en ekstra linje i en databaseforespørgsel.

Hvor det går galt

En løkke sender én forespørgsel pr. række. N+1-mønstret ovenfor ser harmløst ud på en lille udviklingsdatabase og gør en side langsommere i takt med antallet af rækker. Beslut indlæsningen for hver vej gennem koden, der betyder noget, med `selectinload()` eller `joinedload()` i SQLAlchemy og `FETCH_PEERS`, `select_related()` eller `prefetch_related()` i Django. Få derefter det uplanlagte tilfælde til at fejle højlydt: `raiseload()` i SQLAlchemy eller `FETCH_RAISE`, som Djangos dokumentation beskriver som en måde at forhindre utilsigtede forespørgsler i ydelseskritisk kode på.

Ingen læser den SQL, som ORM-biblioteket skriver. En ORM skriver SQL, som ingen på teamet har valgt, og den SQL forbliver usynlig, indtil en side bliver langsom. Djangos optimeringsguide begynder netop her: find ud af, hvilke forespørgsler en side kører, og hvad de koster, brug `QuerySet.explain()` til at se,

hvordan databasen udfører et givet `QuerySet`, og læs `django.db.connection.queries` for at se, hvad Django skriver på kodens vegne. Gør det til en fast del af gennemgangen af enhver vigtig side eller ethvert vigtigt job at læse den SQL, og kør de langsomme sætninger gennem databasens egen forespørgselsplan.

Arbejde, der hører hjemme i databasen, kører i Python. At hente rækker blot for at filtrere dem, sammenligne to af deres felter eller lægge dem sammen flytter hver eneste række over forbindelsen for at gøre i applikationskoden, hvad databasen kunne gøre dér, hvor dataene allerede ligger. Djangos guide nævner alternativerne: `filter()` og `exclude()` til at filtrere i databasen, `F()`-udtryk til at sammenligne felter i samme model og `annotate()` til at aggregere i databasen. Skrivninger følger samme regel, med `bulk_create()` og `bulk_update()` i stedet for ét `save()` pr. objekt, efter at man har tjekket de forbehold, dokumentationen nævner for begge. Ved et meget stort resultatsæt holder et `QuerySet` fast i alt, det har læst, og kan bruge meget hukommelse, og her kan `iterator()` hjælpe.

Den sidste del tvinges gennem ORM-biblioteket. Fowlers eget skøn efterlader 10 til 20 % af mappingproblemerne, og den sidste del kræver ifølge ham altid omhyggeligt arbejde. At bøje en ORM, indtil den producerer en bestemt sætning, gemmer det arbejde inde i genereret SQL, som ingen har valgt. Skriv de sætninger i SQL, og gennemgå dem som al anden kode. Djangos guide nævner trinene nedad: udtrykket `RawSQL` føjer eksplicit SQL til en forespørgsel, mindre portabelt, men mere kraftfuldt, og derudover kan et team skrive sin egen SQL til at hente data eller fylde modeller. Hold ORM-biblioteket til de rutineprægede poster, hvor det sparer mest og koster mindst.

Læs mere

- Catalog of Patterns of Enterprise Application Architecture — Martin Fowlers korte beskrivelser af Active Record, Data Mapper, Unit of Work, Identity Map og Lazy Load.
- ORM Hate — hvorfor objektrelationel mapping er svært, og hvor meget af det et framework kan tage sig af.
- Relationship Loading Techniques — doven indlæsning, select IN, joined og raise i SQLAlchemy 2.1, og hvilken strategi man skal vælge.
- Fetch modes — `FETCH_ONE`, `FETCH_PEERS` og `FETCH_RAISE`, der er nye i Django 6.1.
- Database access optimization — profilering, evaluering af `QuerySet`, arbejde i databasen, bulkmetoder og rå SQL.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er en ORM?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/orm/