Engineer

ORM

et bibliotek som kobler databasetabeller til objekter i kode

Hva det er

En ORM, kort for object-relational mapper, er et bibliotek som kobler tabellene i en relasjonsdatabase til klasser i applikasjonskoden, og radene i dem til objekter. Koden ber om en forfatter og får et `Author` -objekt tilbake, endrer et av attributtene og lagrer det, og ORM-biblioteket skriver `SELECT` -, `INSERT` - og `UPDATE` -setningene som gjør jobben. Modellene i Django og ORM-en i SQLAlchemy, begge skrevet i Python, er to kjente eksempler, og det er dem denne artikkelen bygger på.

Kartleggingen trengs fordi de to sidene organiserer data ulikt. Martin Fowler sier det rett ut i beskrivelsen av mønsteret Data Mapper: objekter og relasjonsdatabaser strukturerer data på hver sin måte, og deler av en objektmodell, som samlinger og arv, har ikke noe motstykke i en relasjonsdatabase. Å flytte data mellom de to blir et problem i seg selv, og en ORM er et ferdig svar på det.

Delene av det svaret fikk navn i Fowlers bok Patterns of Enterprise Application Architecture fra 2003, og mønsterkatalogen fra boken ligger fritt tilgjengelig på nettet. To mønstre beskriver den overordnede formen. Active Record er et objekt som pakker inn én rad i en tabell, selv står for tilgangen til databasen og bærer domenelogikken for de dataene; ifølge designfilosofien til Django følger modellene der nettopp det mønsteret. Data Mapper er et eget lag som flytter data mellom objektene og databasen og holder dem uavhengige av hverandre, slik at objektene ikke trenger å vite at det finnes en database i det hele tatt. Rundt dem ligger flere mindre mønstre. Identity Map sørger for at hvert objekt bare lastes inn én gang. Unit of Work holder en liste over objektene et stykke arbeid har endret, og skriver endringene ut samlet. Lazy Load er et objekt som bare inneholder en del av dataene koden trenger, men vet hvordan resten hentes. Dokumentasjonen til SQLAlchemy nevner to av dem direkte: `Session`

holder innlastede objekter i et identity map, med bare ett objekt per primærnøkkel, og måten den samler endringer og flusher dem til databasen på, er unit of work-mønsteret.

Bruk begrepet i en beslutning

Bruk «Hva er en ORM?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Delen brukerne aldri ser: hvor dataene ligger, hvilke regler som gjelder, og hvordan andre systemer får lov til å snakke med den.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

Den første grunnen er hvor mye kode en ORM sparer. I essayet ORM Hate fra 2012 kaller Fowler problemet med objektrelasjonell mapping vanskelig: det går ut på å holde to svært ulike representasjoner av de samme dataene i takt, og en endring på den ene siden må føres over til den andre. Han anslår at en ORM håndterer omtrent 80 til 90 % av mappingproblemene, og mener at et rammeverk som sparer et team for 80 % av det arbeidet, er verdt å ha, selv om det stopper der. Hver tabell teamet slipper å mappe for hånd, er ett sted mindre der en kolonne kan bli lest inn i feil attributt.

Den andre grunnen er konsistens inne i et kjørende program. Med et identity map returnerer to spørringer som finner den samme raden, det samme objektet, slik at en endring gjort gjennom den ene referansen er synlig gjennom den andre; dokumentasjonen til SQLAlchemy viser det ved å hente den samme primærnøkkelen to ganger og få nøyaktig det samme objektet tilbake. Med en unit of work samler ORM-en ventende innsettinger, oppdateringer og slettinger og skriver dem ifølge funksjonslisten til SQLAlchemy ut i én samlet omgang, slik at applikasjonskoden ikke selv må huske hvilke objekter den har rørt. Django gjør bevisst det motsatte valget: designfilosofien krever et eksplisitt kall til `save()` i stedet for at rammeverket lagrer i det stille, slik at det sendes så få SQL-setninger som mulig.

Den tredje grunnen er hvor grensen går. Fowler beskriver en god ORM som en som tar seg av det kjedelige arbeidet, men gir et mannhull ned til SQL-en når det trengs, og han sier rett ut at den som bygger på en relasjonsdatabase, må vite hvordan en slik database fungerer. Designfilosofien til Django sier det samme fra

rammeverkets side: database-API-et skal vite at det er en snarvei og ikke nødvendigvis hele svaret, og rammeverket skal gjøre det enkelt å skrive egen SQL, enten det er hele setninger eller bare en egen `WHERE` -betingelse. En ORM gjør seg fortjent til plassen sin på den rutinepregede delen av datatilgangen; resten er der et team bruker SQL-kunnskapen sin.

Slik ser det ut i praksis

Kostnaden en ORM oftest skjuler, er antallet spørringer den sender. I SQLAlchemy 2.1 lastes relasjoner lat som standard: de relaterte objektene hentes første gang koden rører attributtet. Dokumentasjonen sier det rett ut: for N innlastede objekter gir tilgang til de latlastede attributtene deres N+1 `SELECT` -setninger. Eksemplet nedenfor forutsetter at klassene `Author` og `Book` allerede er mappet med en en-til-mange-relasjon, `Author.books` , og går gjennom forfatterne på tre måter.

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

# Standard: én SELECT for forfatterne og så én til for hver
# forfatter hvis bøker løkken rører, N+1 setninger til sammen.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN-lasting: en ekstra SELECT henter bøkene til alle
# forfatterne på én gang, med primærnøklene deres 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-lasting: tilgang til author.books gir nå en feil
# i stedet for å sende enda en spørring i det stille.
stmt = select(Author).options(raiseload(Author.books))
```

For en-til-mange- og mange-til-mange-samlinger kaller dokumentasjonen `selectinload()` generelt den beste strategien. `joinedload()` , som den kaller den mest allsidige strategien, legger en `JOIN` til den opprinnelige setningen, som standard en `LEFT OUTER JOIN` , slik at de relaterte radene kommer i det samme resultatsettet. `raiseload()` finnes for å verne mot uønskede innlastinger, og jokertegnformen `raiseload("*")` utvider vernet til de andre relasjonene også, inkludert dem på objektene setningen faktisk henter.

Django 6.1 gir det samme valget et eget navn: fetch modes. Når koden rører et felt som den opprinnelige spørringen ikke hentet, for eksempel en fremmednøkkel, en en-til-en-relasjon eller et felt som er utelatt med `defer()` eller `only()`, henter standardmodusen `FETCH_ONE` det for akkurat den ene instansen, og dokumentasjonen kaller selv resultatet N+1-problemet. `FETCH_PEERS`, som er nytt i 6.1, henter det manglende feltet for instansen og for alle de andre instansene fra det samme `QuerySet`, ut fra antagelsen om at kode som trenger et felt for én rad, som regel trenger det for alle. `FETCH_RAISE` utløser i stedet `FieldFetchBlocked`. En modus settes per `QuerySet`, som i `Book.objects.fetch_mode(models.FETCH_PEERS)`, eller som standard for en modell gjennom en egen manager, og Django kopierer den videre til de relaterte objektene det henter. Der `FETCH_PEERS` ikke passer eller ikke er effektiv nok, viser optimaliseringsguiden til `select_related()` og `prefetch_related()`.

Selskapets egne sider trekker grensen lenger inn mot databasen enn de fleste. Teknologioversikten behandler SQL som et fullverdig språk og ikke noe en ORM genererer, fordi det meste av ytelsesarbeidet selskapet blir hentet inn for, ligger der. I arbeidet hos NextMariner førte grunnleggeren tanken helt ut med et funksjonsbasert datalag i PostgreSQL: hver operasjon appen kan utføre, er én av 1 275 lagrede funksjoner fordelt på 34 skjemaer, og det finnes ingen direkte tabelltilgang fra Go i det hele tatt. Rollen API-et logger inn som, har `EXECUTE` på de funksjonene og `USAGE` på skjemaene deres og ingenting annet, så den kan ikke røre en tabell direkte. Siden legger ikke skjul på prisen: et nytt felt krever en endring i en tabell i tillegg til i en funksjon, ikke bare en ny linje i en spørring.

Hvor det går galt

En løkke sender én spørring per rad. N+1-mønsteret over ser harmløst ut på en liten utviklingsdatabase og gjør en side tregere i takt med antallet rader. Bestem innlastingen for hver vei gjennom koden som betyr noe, med `selectinload()` eller `joinedload()` i SQLAlchemy og `FETCH_PEERS`, `select_related()` eller `prefetch_related()` i Django. Sørg deretter for at det uplanlagte tilfellet feiler høylytt: `raiseload()` i SQLAlchemy, eller `FETCH_RAISE`, som dokumentasjonen til Django beskriver som en måte å hindre utilsiktede spørringer i ytelseskritisk kode på.

Ingen leser SQL-en som ORM-en skriver. En ORM skriver SQL som ingen på teamet har valgt, og den SQL-en forblir usynlig til en side blir treg. Optimaliseringsguiden til Django begynner nettopp her: finn ut hvilke spørringer en side kjører og hva de koster, bruk `QuerySet.explain()` for å se hvordan databasen utfører et gitt `QuerySet`, og les `django.db.connection.queries` for å se

hva Django skriver på vegne av koden. Gjør lesing av den SQL-en til en fast del av gjennomgangen av enhver viktig side eller jobb, og kjør de trege setningene gjennom databasens egen spørringsplan.

Arbeid som hører hjemme i databasen, kjøres i Python. Å hente rader bare for å filtrere dem, sammenligne to av feltene deres eller summere dem flytter hver eneste rad over forbindelsen for å gjøre i applikasjonskoden det databasen kunne gjort der dataene allerede ligger. Guiden til Django lister opp alternativene: `filter()` og `exclude()` for å filtrere i databasen, `F()`-uttrykk for å sammenligne felter i samme modell og `annotate()` for å aggregere i databasen. Skriving følger samme regel, med `bulk_create()` og `bulk_update()` i stedet for én `save()` per objekt, etter at man har sjekket forbeholdene dokumentasjonen nevner for begge. Ved et svært stort resultatsett holder et `QuerySet` på alt det har lest, og kan bruke mye minne, og da kan `iterator()` hjelpe.

Den siste delen tvinges gjennom ORM-en. Fowlers eget anslag etterlater 10 til 20 % av mappingproblemene, og den siste delen krever ifølge ham alltid nøye arbeid. Å bøye en ORM til den produserer en bestemt setning, gjemmer det arbeidet inne i generert SQL som ingen har valgt. Skriv de setningene i SQL, og gjennomgå dem som all annen kode. Guiden til Django nevner trinnene nedover: uttrykket `RawSQL` legger eksplisitt SQL til en spørring, mindre portabelt, men kraftigere, og utover det kan et team skrive sin egen SQL for å hente data eller fylle modeller. Hold ORM-en til de rutinepregede postene, der den sparer mest og koster minst.

Les mer

- Catalog of Patterns of Enterprise Application Architecture — Martin Fowlers korte beskrivelser av Active Record, Data Mapper, Unit of Work, Identity Map og Lazy Load.
- ORM Hate — hvorfor objektrelasjonell mapping er vanskelig, og hvor mye av det et rammeverk kan ta seg av.
- Relationship Loading Techniques — lat lasting, select IN, joined og raise i SQLAlchemy 2.1, og hvilken strategi man bør velge.
- Fetch modes — `FETCH_ONE`, `FETCH_PEERS` og `FETCH_RAISE`, som er nye i Django 6.1.
- Database access optimization — profilering, evaluering av `QuerySet`, arbeid i databasen, bulkmetoder og rå SQL.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er en ORM?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/orm/