Engineer

ORM

Bibliothek, die Datenbanktabellen auf Codeobjekte abbildet

Was es ist

Ein ORM, kurz für Object-Relational Mapper, ist eine Bibliothek, die Tabellen einer relationalen Datenbank auf Klassen im Anwendungscode und Tabellenzeilen auf Objekte abbildet. Code fordert einen Autor an und erhält ein `Author`-Objekt, ändert eines seiner Attribute und speichert es. Der ORM schreibt die dafür nötigen `SELECT`-, `INSERT`- und `UPDATE`-Anweisungen. Die Modelle von Django und der ORM von SQLAlchemy, beide in Python geschrieben, sind zwei bekannte Beispiele, auf die sich dieser Artikel stützt.

Die Abbildung ist nötig, weil beide Seiten Daten unterschiedlich organisieren. Martin Fowler formuliert es in seiner Beschreibung des Data-Mapper-Musters deutlich: Objekte und relationale Datenbanken strukturieren Daten auf unterschiedliche Weise. Teile eines Objektmodells, etwa Sammlungen und Vererbung, haben in einer relationalen Datenbank kein Gegenstück. Daten zwischen beiden zu bewegen wird damit zu einem eigenen Problem. Ein ORM bietet dafür eine fertige Lösung.

Die Bestandteile dieser Lösung wurden in Fowlers 2003 erschienenem Buch Patterns of Enterprise Application Architecture benannt. Sein Musterkatalog ist online verfügbar. Zwei Muster beschreiben den grundlegenden Aufbau. Active Record ist ein Objekt, das eine Tabellenzeile umschließt, seine eigenen Datenbankzugriffe erledigt und die Geschäftslogik dieser Daten trägt. Laut Djangos Entwurfsprinzipien folgen seine Modelle diesem Muster. Data Mapper ist eine getrennte Schicht, die Daten zwischen Objekten und Datenbank bewegt und beide voneinander unabhängig hält. Die Objekte müssen dadurch nichts von der Existenz einer Datenbank wissen. Kleinere Muster ergänzen sie. Identity Map stellt sicher, dass jedes Objekt nur einmal geladen wird. Unit of Work führt eine Liste der durch einen Arbeitsvorgang geänderten Objekte und schreibt die Änderungen gemeinsam. Lazy Load ist ein Objekt, das nur einen Teil der vom Code benötigten Daten hält, aber den Rest abrufen kann. Die SQLAlchemy-

Dokumentation nennt zwei davon direkt: Die `Session` hält geladene Objekte in einer Identity Map, mit genau einem Objekt für jeden Primärschlüssel. Das Sammeln und gemeinsame Schreiben von Änderungen in die Datenbank folgt dem Unit-of-Work-Muster.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein ORM?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Der Teil, den Nutzer nie sehen: wo die Daten liegen, welche Regeln gelten und wie andere Systeme darauf zugreifen dürfen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Warum es wichtig ist

Der erste Grund ist die eingesparte Codemenge. In seinem Essay ORM Hate von 2012 nennt Fowler die objektrelationale Abbildung schwierig: Zwei sehr unterschiedliche Darstellungen derselben Daten müssen übereinstimmen, und eine Änderung auf einer Seite muss auf die andere übertragen werden. Er schätzt, dass ein ORM etwa 80 bis 90 % der Abbildungsprobleme übernimmt. Ein Framework, das einem Team 80 % dieser Arbeit erspart, hält er schon dann für lohnend, wenn es dabei bleibt. Jede Tabelle, die ein Team nicht von Hand abbildet, ist eine Gelegenheit weniger, eine Spalte in das falsche Attribut einzulesen.

Der zweite Grund ist Konsistenz im laufenden Programm. Mit einer Identity Map liefern zwei Abfragen, die dieselbe Zeile finden, dasselbe Objekt zurück. Eine über eine Referenz vorgenommene Änderung ist dadurch auch über die andere sichtbar. Die SQLAlchemy-Dokumentation zeigt dies, indem sie denselben Primärschlüssel zweimal lädt und das identische Objekt erhält. Mit einer Unit of Work sammelt der ORM ausstehende Einfügungen, Änderungen und Löschungen und schreibt sie, wie es die Funktionsübersicht von SQLAlchemy formuliert, gemeinsam in einem Stapel. Der Anwendungscode muss sich deshalb nicht merken, welche Objekte er berührt hat. Django trifft bewusst die gegenteilige Wahl: Seine Entwurfsprinzipien verlangen ein ausdrückliches `save()` statt eines Speicherns im Hintergrund, damit das Framework möglichst wenige SQL-Anweisungen sendet.

Der dritte Grund ist die Lage der Grenze. Fowler beschreibt einen guten ORM als einen, der die Routinearbeit übernimmt und „Zugangsschächte bietet, damit man bei Bedarf hinunter zum SQL gelangt“. Er sagt unmissverständlich, dass jeder, der eine Anwendung auf einer relationalen Datenbank aufbaut, deren Funktionsweise kennen sollte. Djangos Entwurfsprinzipien sagen aus Sicht des Frameworks dasselbe: Die Datenbank-API sollte als Abkürzung statt als vollständige Lösung verstanden werden und das Schreiben von eigenem SQL erleichtern, ob als ganze Anweisung oder als einzelne `WHERE` -Klausel. Ein ORM rechtfertigt seinen Platz beim Großteil der Datenzugriffe, der zur Routine gehört. Für den Rest setzt das Team seine SQL-Kenntnisse ein.

Wie es in der Praxis aussieht

Am häufigsten verbirgt ein ORM die Zahl seiner Abfragen. In SQLAlchemy 2.1 werden Beziehungen standardmäßig verzögert geladen: Die zugehörigen Objekte werden erst beim ersten Zugriff des Codes auf das Attribut abgerufen. Die Dokumentation nennt die Folge ausdrücklich: Bei N geladenen Objekten führt der Zugriff auf deren verzögert geladene Attribute zu N+1 `SELECT` -Anweisungen. Das folgende Beispiel setzt bereits abgebildete Klassen `Author` und `Book` mit einer Eins-zu-viele-Beziehung `Author.books` voraus und durchläuft die Autoren auf drei Arten.

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

# The default: one SELECT for the authors, then one more for each
# author whose books the loop touches, N+1 statements in all.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN loading: a second SELECT fetches the books of every
# author at once, with the authors' primary keys in an IN clause.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise loading: touching author.books now raises an error
# instead of quietly sending another query.
stmt = select(Author).options(raiseload(Author.books))
```

Für Eins-zu-viele- und Viele-zu-viele-Sammlungen nennt die Dokumentation `selectinload()` im Allgemeinen die beste Strategie. `joinedload()`, laut Dokumentation die vielseitigste Strategie, ergänzt die ursprüngliche Anweisung um einen `JOIN`, standardmäßig einen `LEFT OUTER JOIN`. So kommen die zugehörigen Zeilen im selben Ergebnis an. `raiseload()` schützt vor unerwünschtem Nachladen. Seine Platzhalterform `raiseload("*")` erweitert diesen Schutz auf die übrigen Beziehungen, auch auf die der tatsächlich durch die Anweisung geladenen Objekte.

Django 6.1 gibt derselben Entscheidung einen eigenen Namen: Fetch-Modi. Greift Code auf ein Feld zu, das die ursprüngliche Abfrage nicht geladen hat, etwa einen Fremdschlüssel, eine Eins-zu-eins-Beziehung oder ein durch `defer()` oder `only()` ausgelassenes Feld, lädt der Standardmodus `FETCH_ONE` es für genau diese eine Instanz. Die Dokumentation selbst nennt das Ergebnis das N+1-Abfrageproblem. Das in 6.1 neue `FETCH_PEERS` lädt das fehlende Feld für die Instanz und jede weitere Instanz desselben `QuerySet`. Dahinter steht die Überlegung, dass Code ein für eine Zeile benötigtes Feld gewöhnlich auch für alle anderen braucht. `FETCH_RAISE` löst stattdessen `FieldFetchBlocked` aus. Ein Modus wird pro `QuerySet` gesetzt, etwa mit `Book.objects.fetch_mode(models.FETCH_PEERS)`, oder über einen eigenen Manager als Standard eines Modells. Django übernimmt ihn für die abgerufenen zugehörigen Objekte. Wo `FETCH_PEERS` nicht passt oder nicht effizient genug ist, verweist der Optimierungsleitfaden auf `select_related()` und `prefetch_related()`.

Die eigenen Seiten des Unternehmens ziehen die Grenze weiter zur Datenbank hin als die meisten Ansätze. Sein Technologie-Stack behandelt SQL als vollwertige Sprache statt als etwas, das ein ORM generiert, weil dort der Großteil der Performancearbeit liegt, für die das Unternehmen hinzugezogen wird. In ihrer Arbeit unter NextMariner führte die Gründerin diesen Ansatz mit einer funktionsbasierten PostgreSQL-Datenschicht konsequent zu Ende: Jede von der Anwendung ausführbare Operation ist eine von 1.275 gespeicherten Funktionen in 34 Schemas. Direkten Tabellenzugriff aus Go gibt es überhaupt nicht. Die Rolle, mit der sich die API anmeldet, hat `EXECUTE` auf diesen Funktionen und `USAGE` auf ihren Schemas und sonst nichts. Sie kann deshalb keine Tabelle direkt ansprechen. Die Seite benennt den Preis offen: Ein neues Feld bedeutet, sowohl eine Tabelle als auch eine Funktion zu ändern, statt einer Abfrage eine Zeile hinzuzufügen.

Wo es schiefgeht

Eine Schleife sendet eine Abfrage pro Zeile. Das N+1-Muster oben wirkt in einer kleinen Entwicklungsdatenbank harmlos und verlangsamt eine Seite mit wachsender Zeilenzahl. Legen Sie für jeden wichtigen Pfad die Ladestrategie fest: mit `selectinload()` oder `joinedload()` in SQLAlchemy und `FETCH_PEERS`, `select_related()` oder `prefetch_related()` in Django. Lassen Sie den ungeplanten Fall anschließend mit einer deutlichen Fehlermeldung scheitern: mit `raiseload()` in SQLAlchemy oder `FETCH_RAISE`, das Djangos Dokumentation als Möglichkeit beschreibt, unbeabsichtigte Abfragen in performancekritischem Code zu verhindern.

Niemand liest das SQL, das der ORM schreibt. Ein ORM schreibt SQL, das niemand im Team ausgewählt hat. Dieses SQL bleibt unsichtbar, bis eine Seite langsam wird. Djangos Optimierungsleitfaden beginnt hier: Ermitteln Sie, welche Abfragen eine Seite ausführt und was sie kosten. Mit `QuerySet.explain()` sehen Sie, wie die Datenbank ein bestimmtes `QuerySet` ausführt; mit `django.db.connection.queries`, was Django im Auftrag des Codes schreibt. Machen Sie das Lesen dieses SQL zum Bestandteil jedes Reviews wichtiger Seiten oder Jobs und untersuchen Sie langsame Anweisungen mit dem Abfrageplan der Datenbank selbst.

Datenbankarbeit läuft in Python. Wer Zeilen nur lädt, um sie zu filtern, zwei ihrer Felder zu vergleichen oder sie zu summieren, überträgt jede Zeile über die Verbindung. Der Anwendungscode erledigt damit etwas, das die Datenbank dort tun könnte, wo die Daten bereits liegen. Djangos Leitfaden nennt die Alternativen: `filter()` und `exclude()` filtern in der Datenbank, `F()`-Ausdrücke vergleichen Felder desselben Modells, und `annotate()` aggregiert in der Datenbank. Für Schreibzugriffe gilt dasselbe: `bulk_create()` und `bulk_update()` ersetzen ein `save()` pro Objekt, nachdem die in der Dokumentation für beide genannten Einschränkungen geprüft wurden. Bei einer sehr großen Ergebnismenge hält ein `QuerySet` alles Gelesene fest und kann viel Speicher beanspruchen. `iterator()` kann helfen.

Der letzte Teil wird durch den ORM gezwungen. Fowlers eigene Schätzung lässt 10 bis 20 % der Abbildungsprobleme übrig. Dieser letzte Teil erfordert nach seinen Worten immer sorgfältige Arbeit. Den ORM so lange zu verbiegen, bis er eine bestimmte Anweisung erzeugt, versteckt diese Arbeit in generiertem SQL, das niemand gewählt hat. Schreiben Sie solche Anweisungen in SQL und prüfen Sie sie wie anderen Code. Djangos Leitfaden nennt die Übergänge zu dieser Ebene: Der Ausdruck `RawSQL` ergänzt eine Abfrage um ausdrückliches SQL,

weniger portabel, aber leistungsfähiger. Darüber hinaus kann ein Team eigenes SQL zum Abrufen von Daten oder Befüllen von Modellen schreiben. Behalten Sie den ORM für die Routinefälle, bei denen er am meisten spart und am wenigsten kostet.

Weiterführende Lektüre

- Musterkatalog zu Patterns of Enterprise Application Architecture — Martin Fowlers Zusammenfassungen von Active Record, Data Mapper, Unit of Work, Identity Map und Lazy Load.
- ORM Hate — weshalb die objektrelationale Abbildung schwierig ist und wie viel davon ein Framework übernehmen kann.
- Techniken zum Laden von Beziehungen — verzögertes Laden, Select-IN-, Join- und Raise-Laden in SQLAlchemy 2.1 und die Wahl der geeigneten Strategie.
- Fetch-Modi — `FETCH_ONE`, `FETCH_PEERS` und `FETCH_RAISE`, neu in Django 6.1.
- Optimierung von Datenbankzugriffen — Profiling, Auswertung von `QuerySet`, Arbeit in der Datenbank, Stapelmethoden und direktes SQL.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein ORM?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/orm/