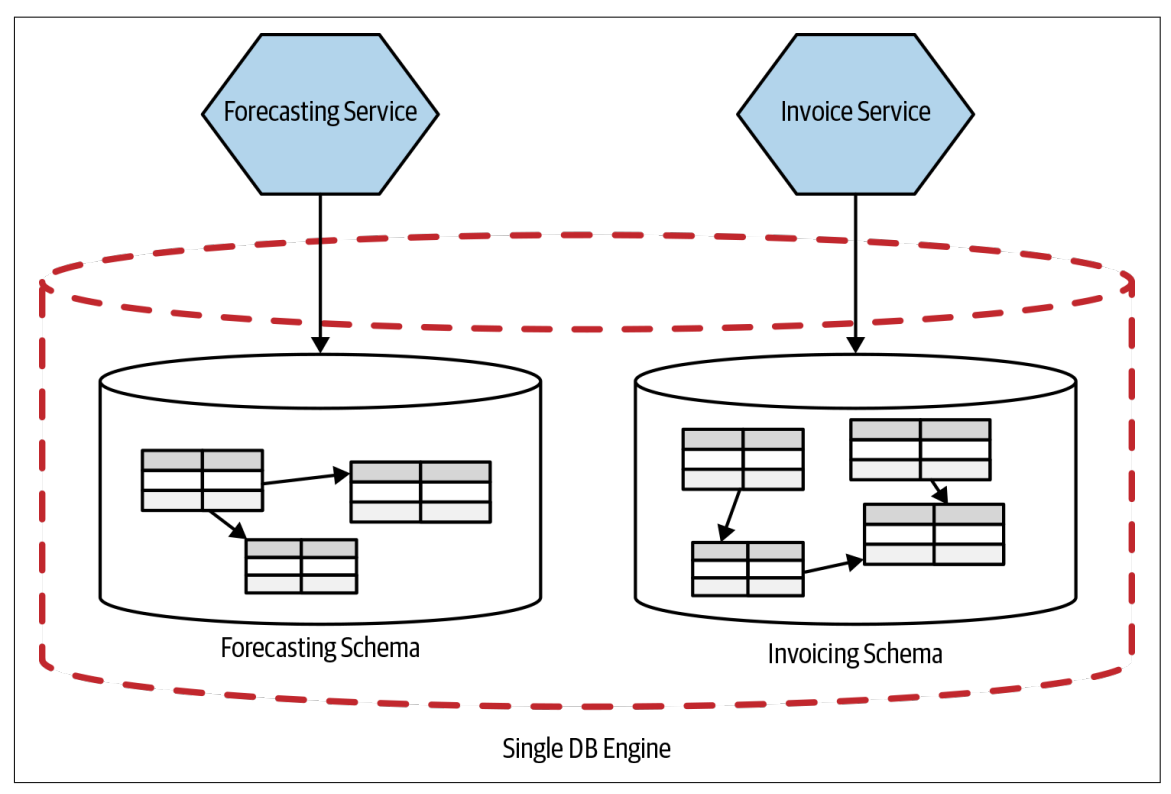
**ІНДЗ № 9**

**Розділення бази даних (сторінки 158-170)**

Фізичне та логічне розділення баз даних

Коли говоримо про поділ наших баз даних у цьому контексті, ми насамперед намагаємося досягти логічного розділення. Як ми бачимо на малюнку 4-24, єдиний механізм бази даних цілком здатний розміщувати більше однієї логічно-розділеної схеми.



*Малюнок 4-24. Дві служби, які використовують окремі логічні схеми, обидві працюють на одній фізичній базі даних*

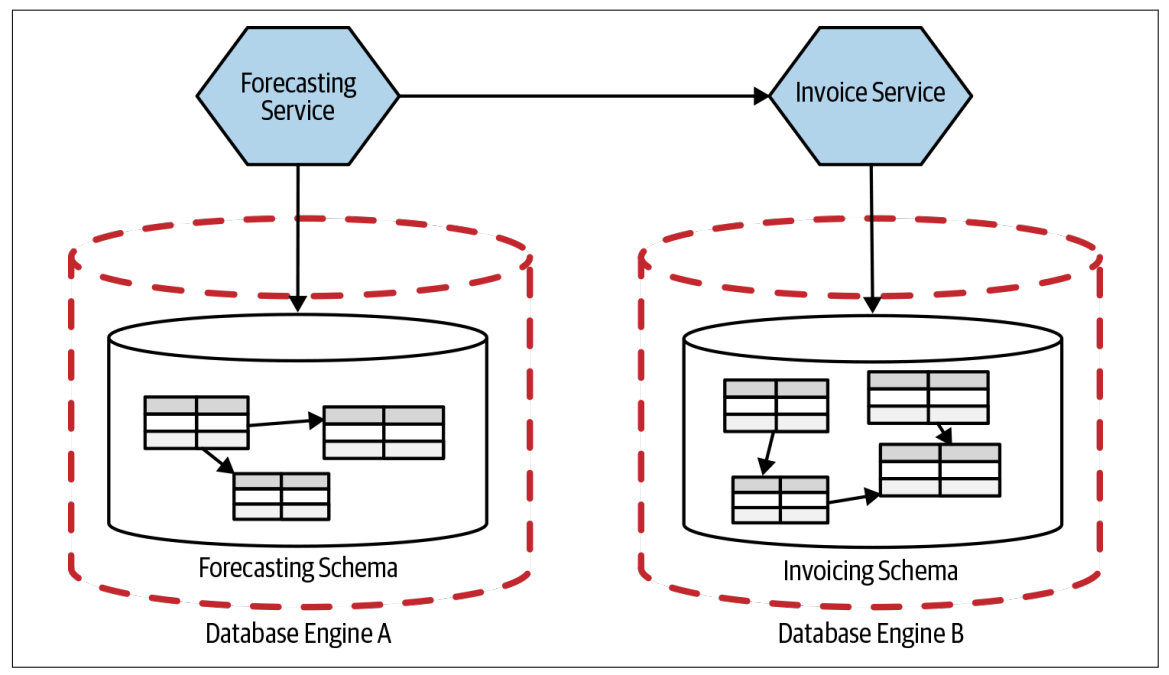
Ми могли б піти далі й розмістити кожну логічну схему також на окремих механізмах баз даних, що також дало б нам фізичне розділення, як ми бачимо на малюнку 4-25.

Логічна та фізична декомпозиція досягають різних цілей. Логічна декомпозиція дозволяє здійснювати простішу незалежну зміну і приховувати інформаціїю, тоді як фізичне розкладання потенційно покращує надійність системи та може допомогти усунути суперечку за ресурси, дозволяючи покращити пропускну здатність або затримку.

Коли ми логічно розкладаємо схеми нашої бази даних, але зберігаємо їх на тій самій фізичній базі даних, як на малюнку 4-24, у нас є потенційна єдина точка відмови.

Якщо механізм бази даних вимикається, це впливає на обидві служби. Однак світ не такий простий. Багато механізмів баз даних мають механізми для уникнення окремих точок збою, такі як багатоосновні режими баз даних, механізми гарячого відновлення після відмови тощо.

Насправді, можливо, було докладено значних зусиль для створення високостійкого кластера бази даних у вашій організації, і може бути важко виправдати наявність кількох кластерів через час, зусилля та кошти, які можуть бути залучені.



*Малюнок 4-25. Дві служби, які використовують окремі логічні схеми, кожна з яких працює на власному фізичному механізмі бази даних*

Інше зауваження полягає в тому, що наявність кількох схем, які спільно використовують один і той самий механізм бази даних, може знадобитися, якщо ви хочете відкрити представлення вашої бази даних. Як вихідна база даних, так і схеми, що містять представлення, можуть знадобитися розташувати в одній базі даних.

Щоб була можливість запускати окремі служби на різних фізичних механізмах баз даних, потрібно вже логічно розкласти їхні схеми!

Спочатку розділити базу даних чи код?

Видобуток мікросервісу не буде «завершено», доки код програми не буде запущено у його власній службі, а дані, якими він керує, не будуть витягнуті у його власну логічно ізольовану базу даних. Як це вилучення повинно бути послідовним? Є кілька варіантів:

• Спочатку розділити базу даних, а потім код.

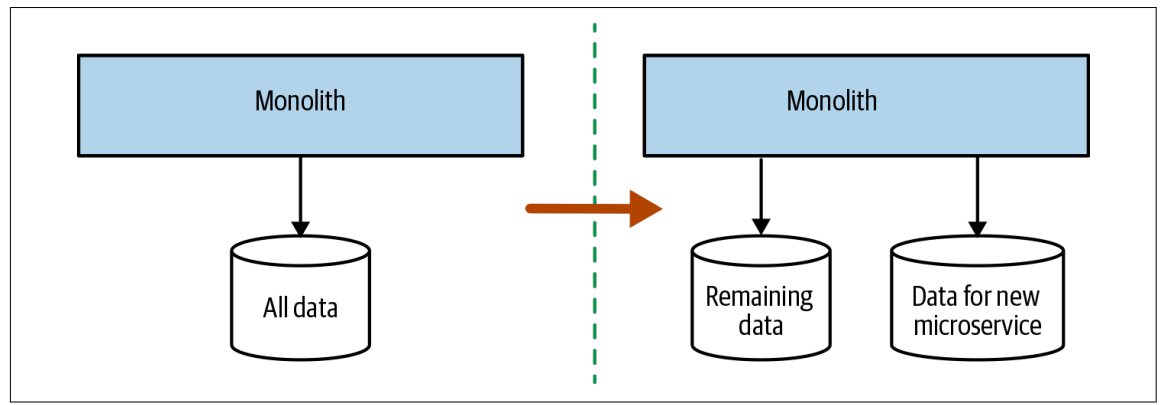
• Спочатку розділити код, а потім базу даних.

• Розділити їх обох одночасно.

Кожен має свої плюси і мінуси. Розглянемо ці параметри разом із деякими шаблонами, які можуть допомогти залежно від вашого підходу.

**Спочатку розділити базу даних**

За допомогою окремої схеми потенційно збільшимо кількість викликів бази даних для виконання однієї дії. Раніше ми могли мати всі потрібні дані в одному операторі SELECT, тепер може знадобитися отримати дані з двох місць і об’єднати в пам’яті. Але ми порушуємо цілісність транзакцій, коли переходимо до двох схем, що може мати значний вплив на наші програми. Розділивши схеми, але зберігаючи разом код програми, як показано на малюнку 4-26, ми маємо можливість скасувати внесені зміни або продовжувати налаштовувати речі, не впливаючи на споживачів наших послуг. Якщо переконаємося, що поділ БД має сенс, можемо подумати про поділ коду програми на дві служби.



*Малюнок 4-26. Спершу розділивши схему, ви можете раніше помітити проблеми з продуктивністю та цілісністю транзакцій*

Зворотний бік полягає в тому, що такий підхід навряд чи принесе значну короткострокову вигоду. З цієї причини піти цим шляхом можна лише якщо особливо турбують потенційні проблеми з продуктивністю або узгодженістю даних. Також потрібно враховувати, що якщо сам моноліт є системою чорної скриньки, як частина комерційного програмного забезпечення, ця опція для нас недоступна.

Примітка щодо інструментів

Змінювати бази даних важко з багатьох причин, одна з яких полягає в тому, що доступні обмежені інструменти, які дозволяють нам легко вносити зміни. Завдяки коду ми маємо інструменти рефакторингу, вбудовані в наші IDE, і ми маємо додаткову перевагу, оскільки системи, які ми змінюємо, принципово не мають стану. З базою даних речі, які ми змінюємо, мають стан, але нам також бракує хороших інструментів для рефакторингу.

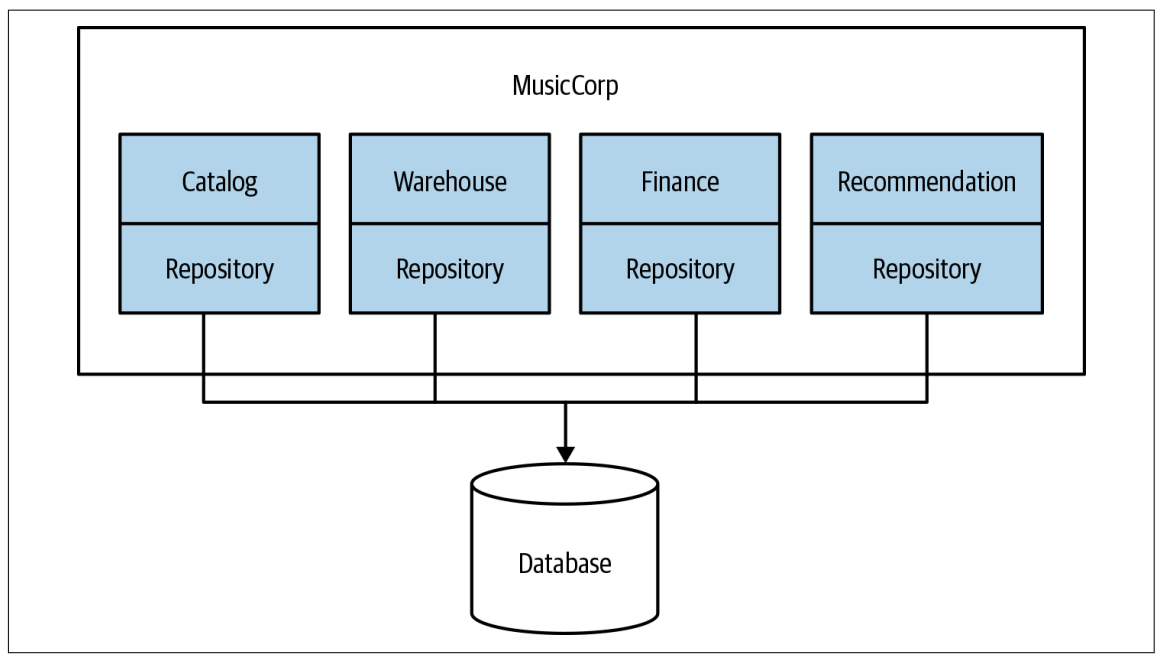
Ніка Ешлі та Грема Таклі створили інструмент із відкритим кодом під назвою DBDeploy. Він працював, дозволяючи фіксувати зміни в сценаріях SQL, які можна було запускати детермінованим способом на схемах. Кожна схема мала спеціальну таблицю, яка використовувалася для відстеження того, які сценарії схеми були застосовані.

Мета DBDeploy полягала в тому, щоб дозволити вам вносити поетапні зміни в схему, контролювати кожну версію змін і дозволити виконувати зміни в кількох схемах у різний час (схеми для розробників, тестування та виробництва).

Шаблон: сховище для обмеженого контексту

Поширеною практикою є наявність рівня сховища, який підтримується певним фреймворком, таким як Hibernate, щоб прив’язувати ваш код до бази даних, що полегшує відображення об’єктів або структур даних у базах даних та з бази даних. Замість того, щоб мати єдиний рівень репозиторію для всіх проблем із доступом до даних, корисно розбити ці сховища за лініями обмежених контекстів, як показано на малюнку 4-27.

Наявність коду відображення бази даних, розміщеного всередині коду для заданого контексту, може допомогти зрозуміти, які частини бази даних використовуються в яких бітах коду. Наприклад, Hibernate може зробити це дуже чітким, якщо ви використовуєте щось на зразок файлу відображення для обмеженого контексту. Таким чином, ми можемо побачити, які обмежені контексти мають доступ до яких таблиць у нашій схемі. Це може допомогти зрозуміти, які таблиці потрібно перемістити в рамках будь-якої майбутньої декомпозиції.



*Малюнок 4-27. Розділення шарів сховища*

Однак це не дає всієї історії. Наприклад, можна сказати, що фінансовий код використовує таблицю бухгалтерської книги, а код каталогу використовує таблицю позицій, але може бути незрозуміло, що база даних забезпечує зв’язок зовнішнього ключа від таблиці книги до таблиця позицій. Щоб побачити ці обмеження на рівні бази даних, потрібно використовувати інший інструмент для візуалізації даних. Для початку можна скористатися таким інструментом SchemaSpy, який може генерувати графічні представлення зв’язків між таблицями.

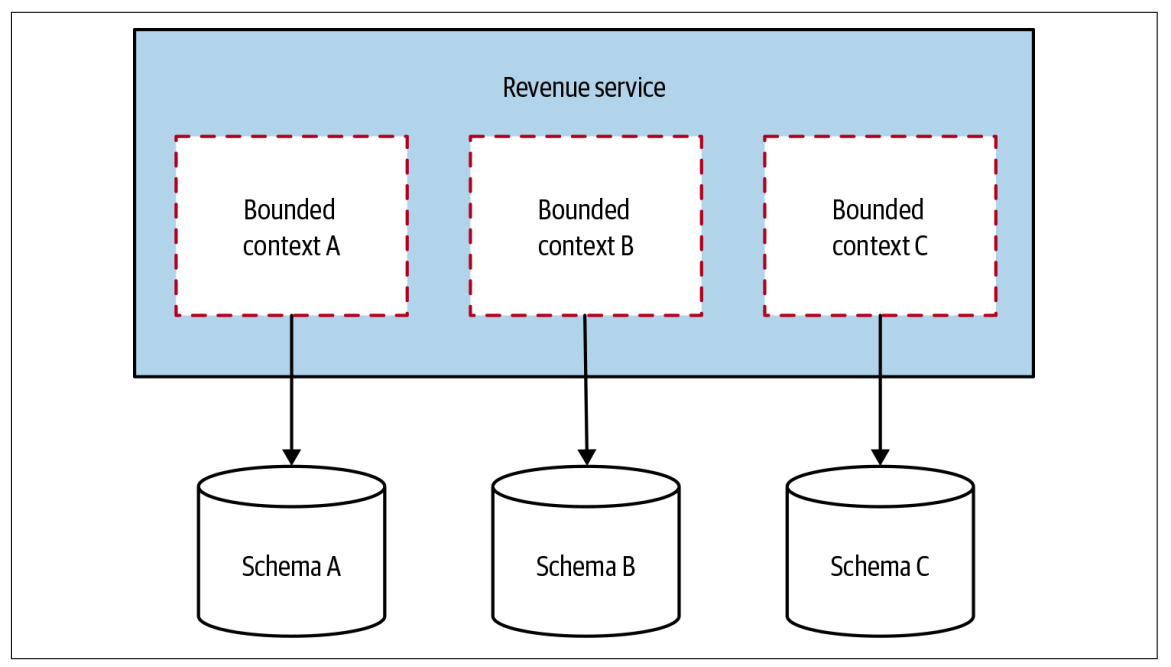
Де це використовувати

Цей шаблон дійсно ефективний у будь-якій ситуації, коли ви хочете переробити моноліт, щоб краще зрозуміти, як його розділити. Розбивання цих рівнів сховища за принципами домену значною мірою допоможе зрозуміти, де можуть існувати шви для мікросервісів не лише у вашій базі даних, але й у самому коді.

Шаблон: База даних на обмежений контекст

Після того як чітко відокремили доступ до даних з точки зору програми, можемо продовжити цей підхід у схемі. Головним в ідеї незалежного розгортання мікросервісів є той факт, що вони повинні володіти власними даними. Перш ніж приступити до виділення коду програми, можемо розпочати цю декомпозицію, чітко відокремивши наші бази даних навколо ліній наших визначених обмежених контекстів.

У ThoughtWorks ми впроваджували деякі нові механізми для розрахунку та прогнозування прибутку компанії. Ми визначили три широкі області функціональності, які потрібно було написати. Функціональність здається досить окремою, але є занепокоєння щодо додаткової роботи, яку принесе ця функціональність, якщо її розмістити в окремих мікросервісах. Та зупинилися на моделі, у якій нова функціональність прибутку була фактично розгорнута як єдиний сервіс, що містить три ізольовані обмежені контексти (кожен з яких закінчився як окремі файли JAR), як показано на малюнку 4-28.



*Малюнок 4-28. Кожен обмежений контекст у службі доходів мав власну окрему схему бази даних, що дозволяло відокремити пізніше*