Modeling Microservices

Hönnun og smíði hugbúnaðar

Haust 2022

Hvað gerir góð service boundary?

- Viljum miða á service sem er **independently changed, deployed and released**

- Mikið af fræði tengt modular hugbúnað nýtist (t.d. OOP)
  - Microservices er líka form af *modular decomposition*
  - Þrjár lykil hugmyndir → **Information Hiding, Cohesion, Coupling**

Information Hiding

- Viljum fela smáatriðin eins mikið og mögulega
- Ein besta leiðin til að fá það mesta úr modular architecture
- Minnkar coupling við client-a
- Viljum geta breytt virkni án þess að brjóta client-a
  - Viljum hafa Independent deployability
- Náum þessu með velskilgreindum APIs
  - Event schema, REST, gRPC, graphQL…
  - Felum innri virkni
  - Felum innri gögn
  - Ráðum hvaða virkni/gögn eru birt og **hvernig**

Information Hiding Frh.

The connections between modules are the assumptions which the modules make about each other.

Cohesion

- *the code that changes together, stays together*
- Óskyldur domain kóði á heima annars staðar
- Skyldur domain kóði á heima saman
- Viljum hafa hátt cohesion innan service
- Viljum bara þurfa að gera breytingar í einu service
  - Viljum hafa Independent deployability
  - Tímafrekt að breyta mörgum service

Loose Coupling

- Eitt af helstu hvötum bakvið microservices
- Microservices eiga að vera decoupled frá hvort öðrum
- Services eiga að vita eins lítið um hvort annað og mögulega
- Breyting í einu service ætti ekki að hafa áhrif á annað
  - Viljum hafa Independent deployability
- Okkur tekst þetta með vel skilgreindum APIs
  - **Information hiding**
  - Verndar clients frá breytingu á gögnum og virkni
  - Events gefa okkur sérstaklega decoupled services
  - Client libraries gefa okkur sérstaklega coupled services

Coupling er óumflýjanleg

- Munum alltaf hafa coupling
- Skyldur domain kóði mun alltaf vera coupled saman
- Coupling þarf ekki að vera slæmur hlutur
  - Viljum bara meðhöndla coupling þannig breytingar verða auðveldar
- Viljum hafa hátt coupling innan service og lágt á milli service-a

Coupling og Cohesion

- Skyld hugtök
  - Lýsa bæði sambandi milli hluta
- Dreifð skyld domain logic → hátt external service coupling
  - Breyting leiðir til breytinga í mörgum service
- Hátt internal service cohesion → Hátt internal service coupling
  - Þetta er það sem við viljum!
  - Skyldur og coupled kóði á heima saman
- *A structure is stable if cohesion is strong and coupling is low*

Tegundir af coupling á milli service-a

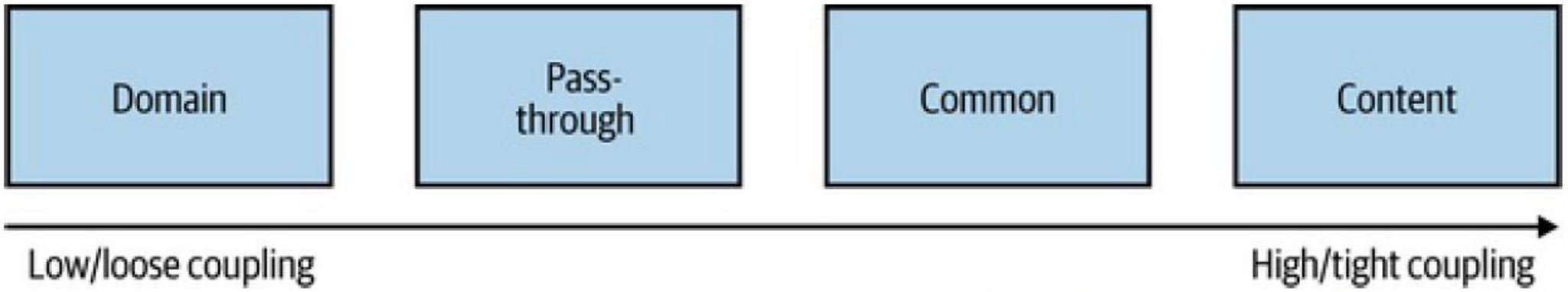


Figure 2-1. The different types of coupling, from loose (low) to tight (high)

Domain Coupling

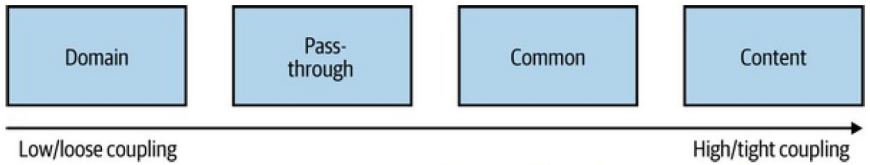


Figure 2-1. The different types of coupling, from loose (low) to tight (high)

- Þegar eitt microservice talar við annað
  - Þarf gögn eða virkni
- Óumflýjanleg coupling
  - Services þurfa að tala saman
- Viljum lágmarka þessa coupling
  - Information hiding
  - Vel skilgreindir APIs
  - Forðast að tala við of mörg service
  - Forðast *chatty communication*
  - **Mikilvægt að service boundary eru rétt**

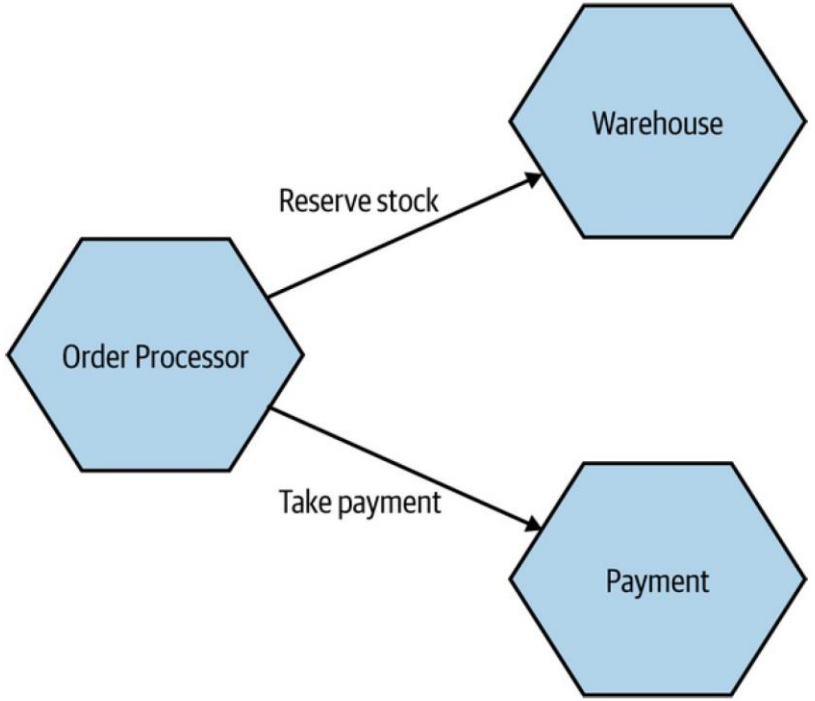


Figure 2-2. An example of domain coupling, where `Order Processor` *needs to make use of the functionality provided by other microservices*

Temporal Coupling

- Þegar eitt service krefst að annað service er keyrandi á sama tíma
- Hægt að koma í veg fyrir með events
- Oft óumflýjanleg coupling

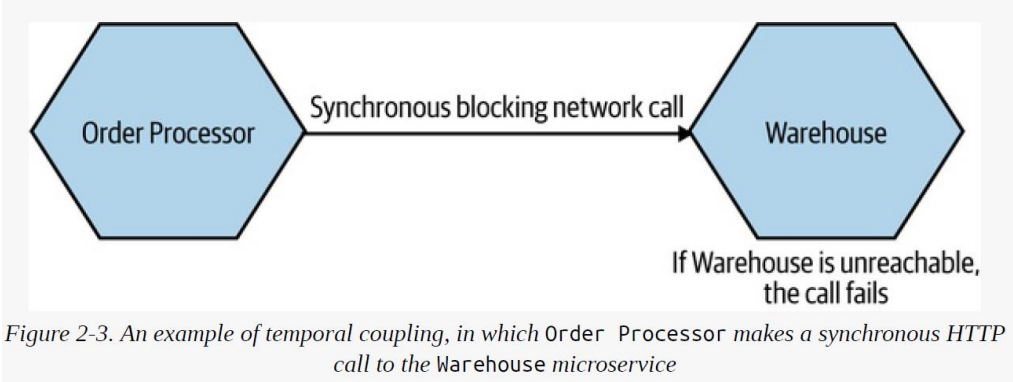


Figure 2-3. An example of temporal coupling, in which `Order Processor` *makes a synchronous HTTP call to the* `Warehouse` *microservice*

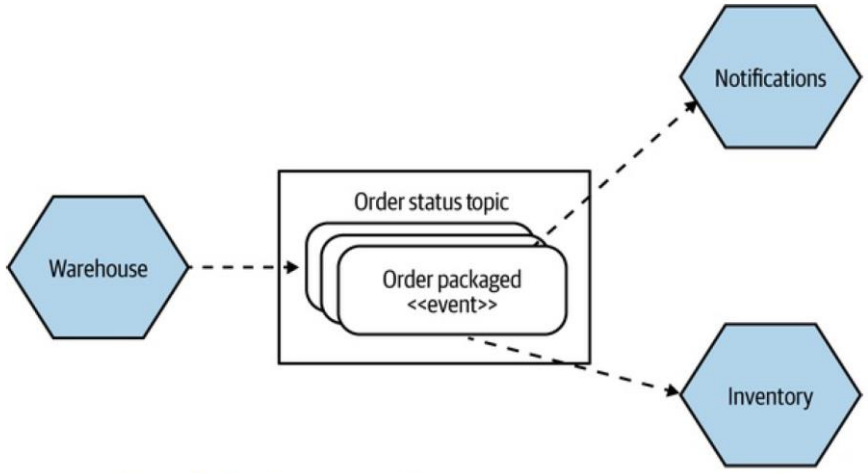


Figure 4-11. The `Warehouse` *emits events that some downstream microservices subscribe to*

Pass-Through Coupling

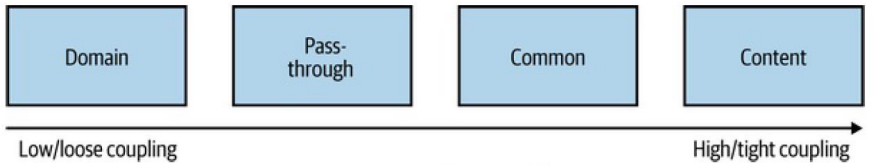
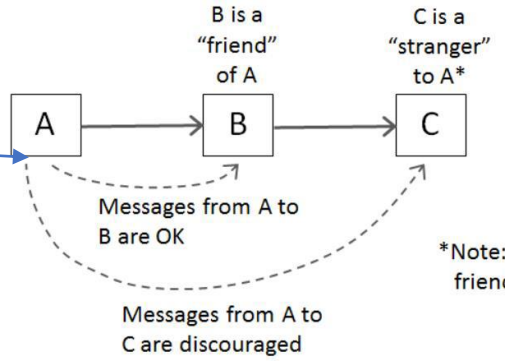
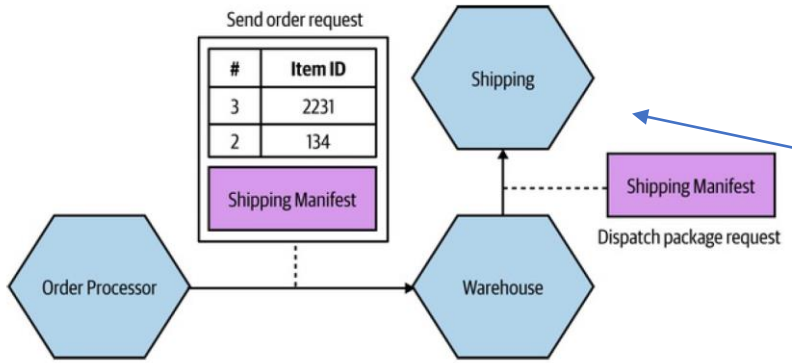


Figure 2-1. The different types of coupling, from loose (low) to tight (high)

- Þegar service ***b*** sendir gögn frá service ***c*** áfram til service ***a***
- Service ***a*** coupled við service ***b*** og núna einnig við ***c***
- Sama vandamál og Law of Demeter leysir





Ef shipping þarf önnur gögn þá þurfa bæði warehouse og order processor að breytast

Figure 2-4. Pass-through coupling, in which data is passed to a microservice purely because another downstream service needs it

Pass-Through Coupling Lausn 1

Order Processor talar bæði við Warehouse og Shipping

Domain Coupling > Pass-through Coupling (almennt séð)

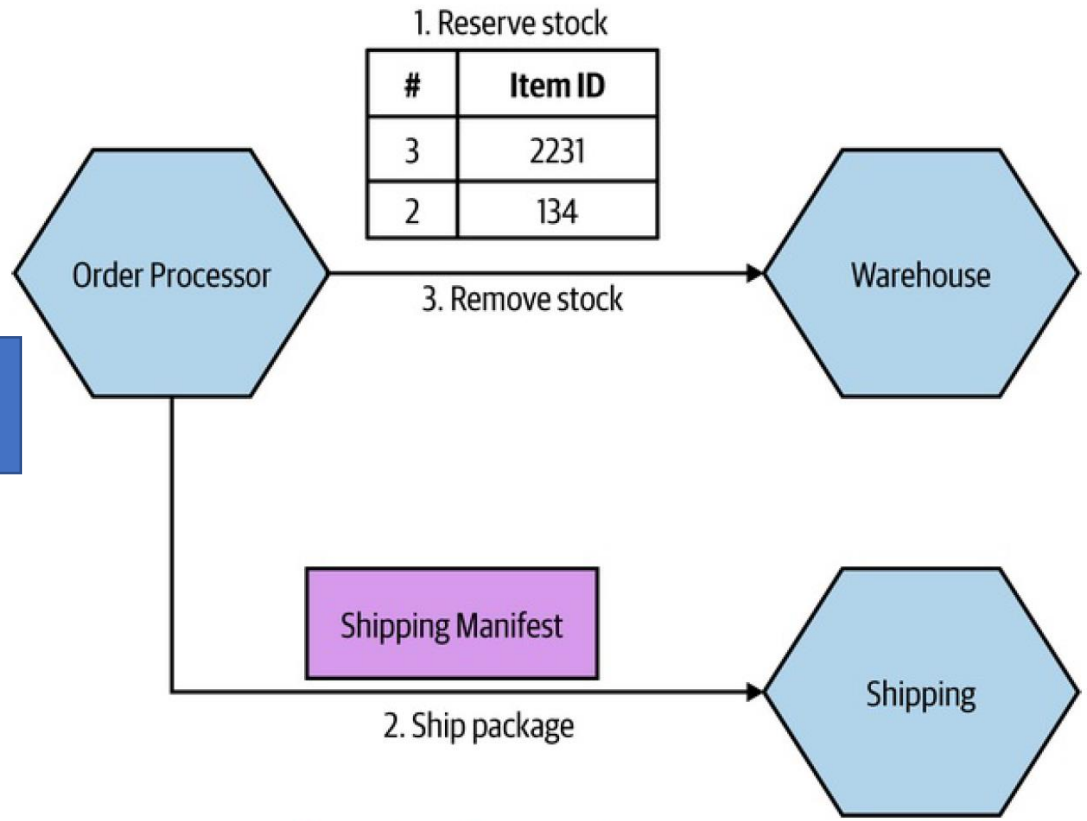


Figure 2-5. One way to work around pass-through coupling involves communicating directly with the downstream service

Vandamál:

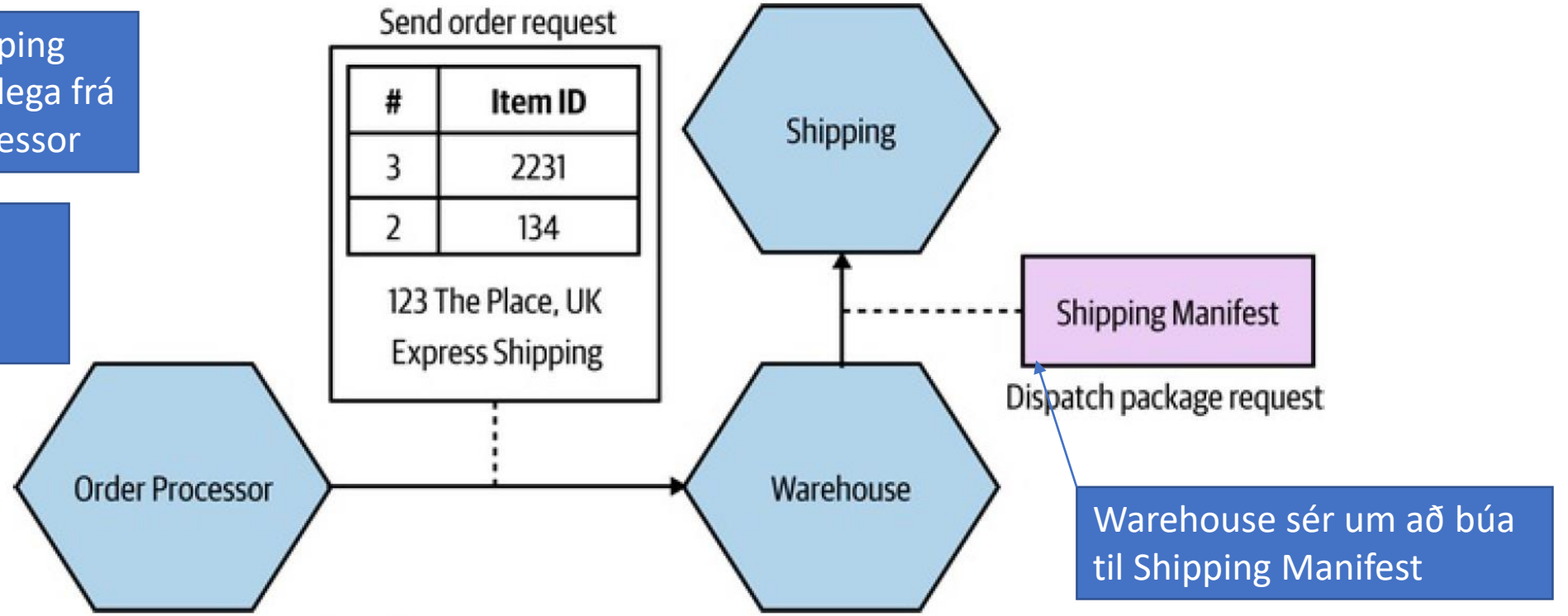
- Meiri domain coupling í order processor
- Meira flækjustig / logic í order processor

Pass-Through Coupling Lausn 2

Felum shipping logic algjörlega frá Order Processor

Færum lower level detail í lower level component-a

Order Processor **orðinn decoupled frá shipping service**



Warehouse sér um að búa til Shipping Manifest

Figure 2-6. Hiding the need for a `Shipping Manifest` *from the* `Order Processor`

Common Coupling

- Þegar tvö (eða fleiri) service nota sömu gögn
  - T.d. tvö service að nota sama gagnagrunn
- Breyting á strúktúr gagna hefur áhrif á mörg service
- Minna vandamál þegar gögn og strúktur breytast sjaldan
  - T.d. einhver static gögn
- Vandamál þegar gögn og strúktur breytast oft
- Vandamál þegar tvö service vista í sömu gögn
- Getur leitt til performance vandamála

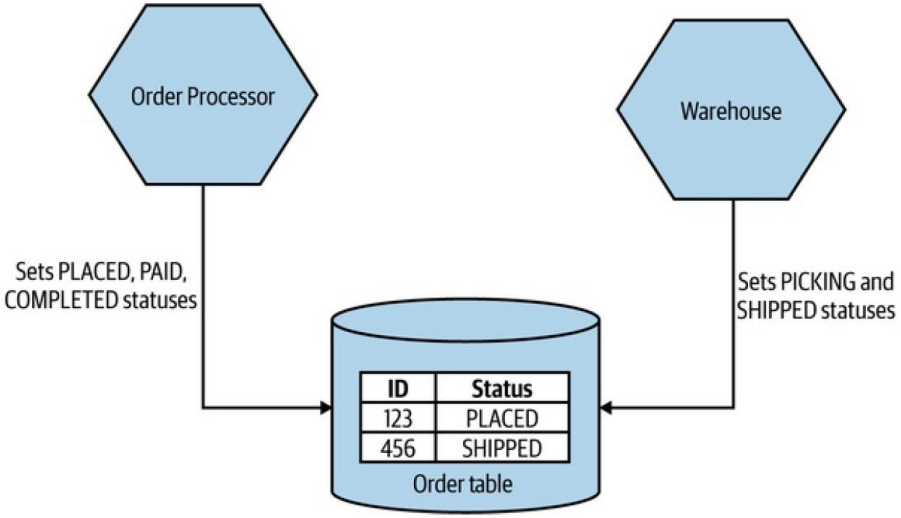


Figure 2-8. An example of common coupling in which both `Order Processor` *and* `Warehouse` *are updating the same order record*

Common Coupling vandamál

Getum við verið viss um að eitt service yfirskrifar ekki gögn annars?

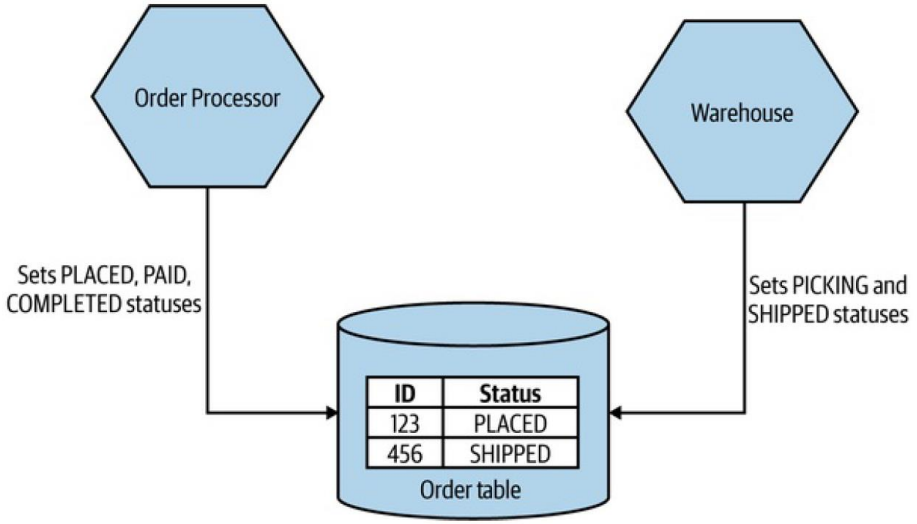


Figure 2-8. An example of common coupling in which both `Order Processor` *and* `Warehouse` *are updating the same order record*

Bæði Order Processor og Warehouse **hafa ábyrgð og eign** yfir Order gögnunum

Getum við verið viss um að eitt service brýtur ekki heimsmynd annars?

Bæði Order Processor og Warehouse sjá um logic í þessu state machine **(of lítið cohesion)**

Getum við verið viss um að consistency og state helst rétt?

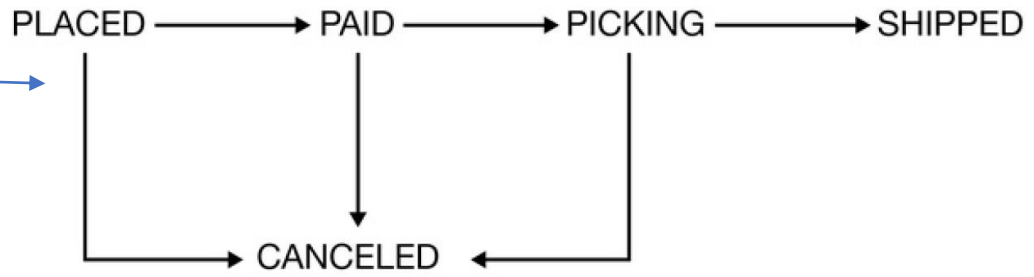


Figure 2-9. An overview of the allowable state transitions for an order in MusicCorp

Common Coupling Lausn

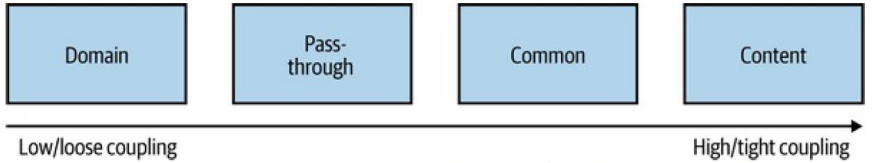


Figure 2-1. The different types of coupling, from loose (low) to tight (high)

Order Processor og Warehouse þurfa að senda request um að sækja / breyta stöðu á gögnum

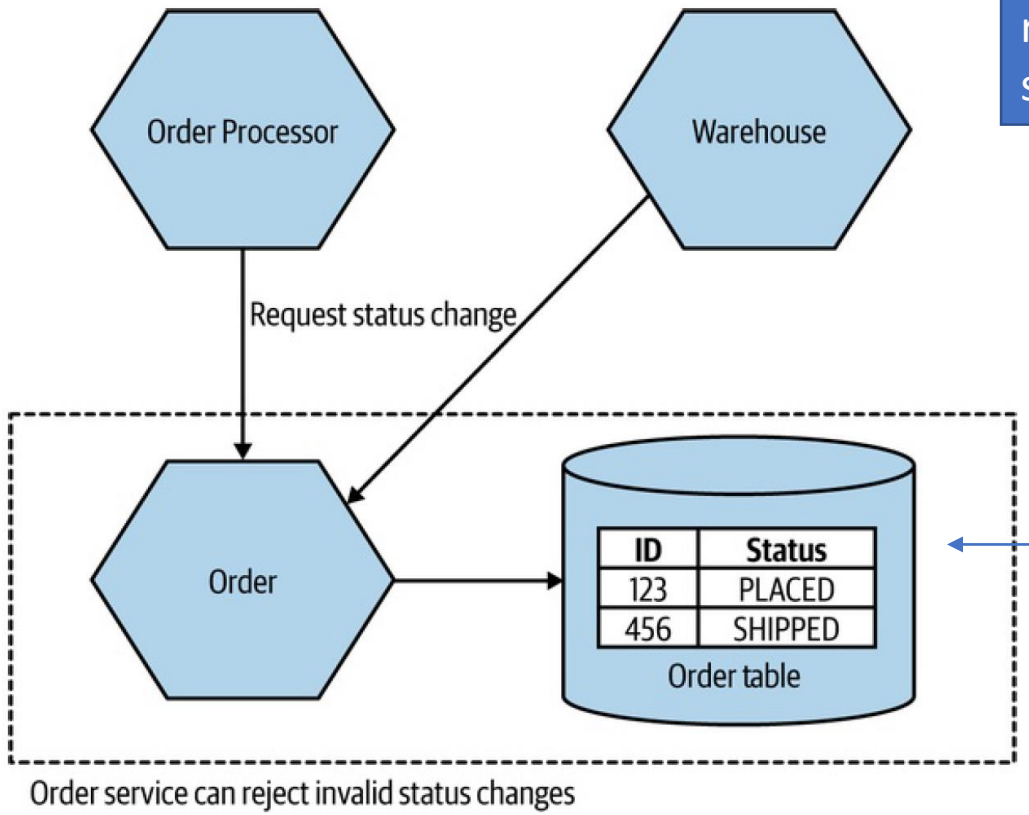


Figure 2-10. Both `Order Processor` and `Warehouse` can request that changes be made to an order, but the `Order` microservice decides which requests are acceptable

Order service hefur fullt vald, eign og ábyrgð yfir gögnunum

Order Service sér um að gögn eru consistent og í réttu state-i

Order Service má hafna requests

Common Coupling Lausn frh.

Ef order Service væri bara þunnur CRUD wrapper þá værum við ennþá með sama vandamál

WARNING

If you see a microservice that just looks like a thin wrapper around database CRUD operations, that is a sign that you may have weak cohesion and tighter coupling, as logic that should be in that service to manage the data is instead spread elsewhere in your system.

Content Coupling

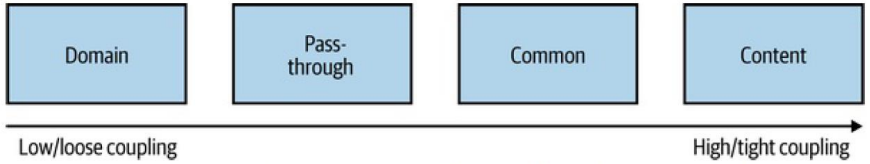


Figure 2-1. The different types of coupling, from loose (low) to tight (high)

- Þegar eitt service teygir sig í / breytir gögnum hjá öðru service
- Munur á Common Coupling
  - Ekki sameiginleg gögn
  - Gögnin eru í eigu annars service
  - Í Common Coupling er skilningur að gögnin eru í eigu margra
- Breytingar á strúktur og gögnum líklegar til að brjóta virkni
- Erfitt að vita hvort staða gagna er rétt
  - Erfiðara en með common coupling

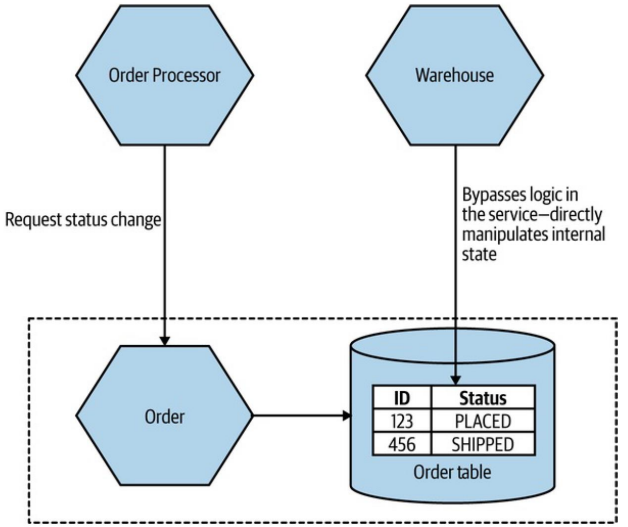


Figure 2-11. An example of content coupling in which the `Warehouse` is directly accessing the internal data of the `Order` service

Content Coupling Vandamál

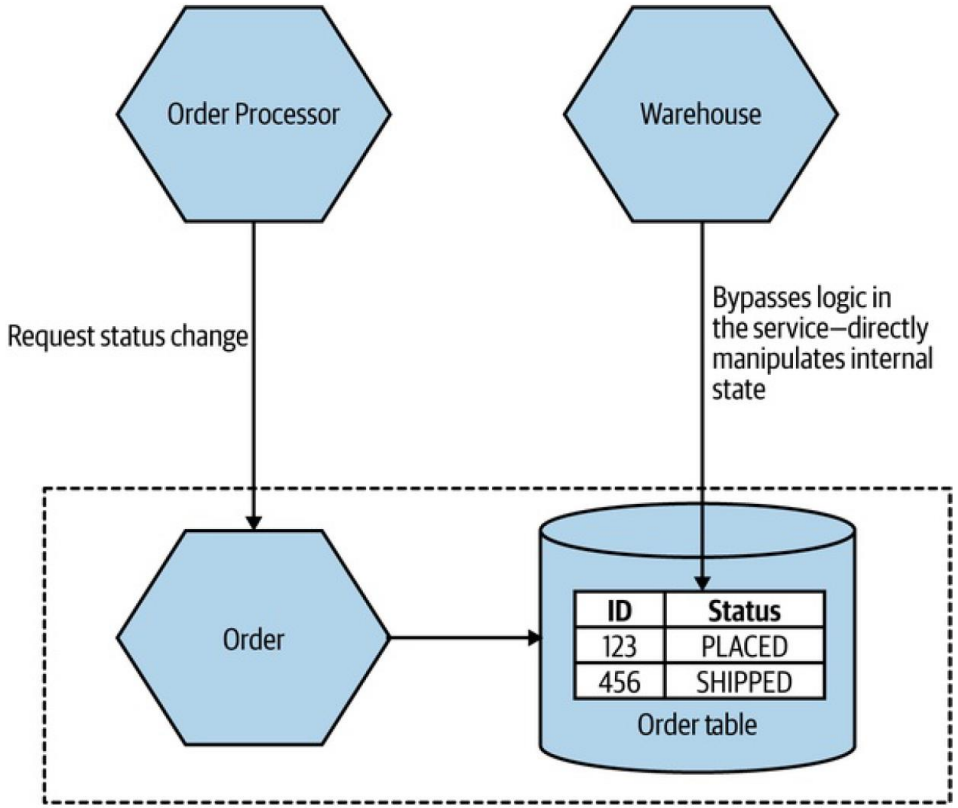


Figure 2-11. An example of content coupling in which the `Warehouse` *is directly accessing the internal data of the* `Order` *service*

Warehouse fer framhjá reglum / logic / validation í Order Service

Getur leitt til inconsistent eða **rangt** state

Strúktur breytingar brjóta warehouse auðveldlega

Domain Driven Design (DDD)

- Helsta leið til að finna service boundary er í kringum domain-ið sjálf
  - Vertical sliced architecture
- DDD er hugmyndafræði mótað af Eric Evan's (Domain-Driven Design)
- DDD hjálpar okkur að skilja og represent-a domain-ið betur
- DDD setur business domain-ið í hjarta hugbúnaðarins
  - Breytingar koma oftast fram í formi krafa frá business domain-inu
- DDD þætti sem við munum skoða
  - Ubiquitous Language
  - Aggregate
  - Bounded Context

Ubiquitous Language

- Hugmyndin um að nota sömu hugtök í kóða sem domain experts nota
- Auðveldar að model-a kerfið eins og raunheiminn
- Auðveldar samskipti og samstarf við domain experts
- Auðveldara að vinna requirements
- Auðveldar að kerfið er eins og það á að vera
- Auðveldar að skilja kóðann
- Stundum vandamál þegar domain experts nota annað tungumál en ensku

Aggregates

- High Level Domain hlutur sem vefur aðra domain hluti
  - Life cycle-ið er í kringum high level aggregate hlutinn
  - Myndar grouping / hjúpun af hlutum
  - Undirliggjandi hlutir hafa tilgang eingöngu sem hluti af aggregate-inu
- Aggregate oft minnsta einnig til að framkvæma aðgerðir á
  - Tryggir að state á undirliggjandi hlutum er rétt
  - *Self-contained state machine*
- T.d. Order Aggregate
  - Inniheldur margar vörulínur
  - Myndum kannski eingöngu framkvæma aðgerðir á *order-basis*
- Aggregate ætti ekki að vera dreift á milli service-a
- Eitt service getur haft mörg aggregate

Aggregate Sambönd

- Aggregates geta haft sambönd sín á milli

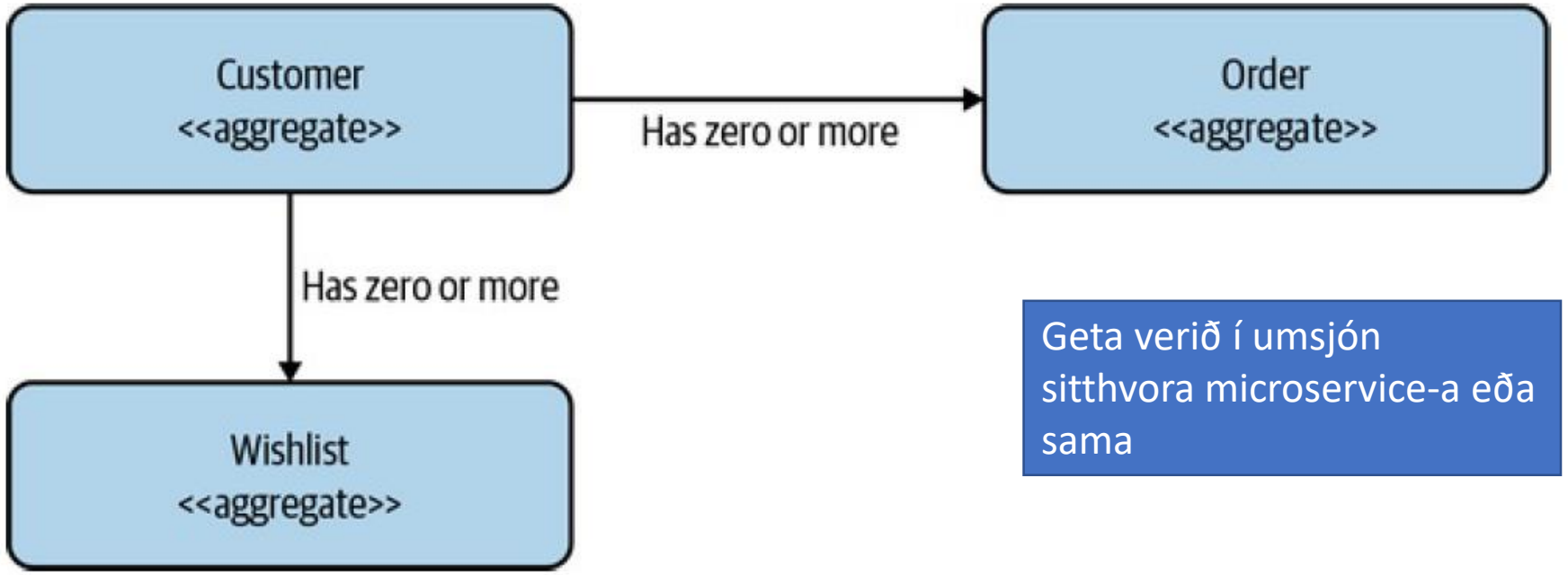


Figure 2-12. One Customer aggregate may be associated with one or more Order or Wishlist aggregates

Bounded Context

- Ákveðið Organizational boundary / Domain boundary
- Fela innri virkni
- Innihalda eitt eða fleiri skyld aggregate
- Bounded Contexts mynda service scopes
  - Hægt að brjóta seinna upp eftir aggregates
  - Microservices draga innblástur úr DDD og bounded context hugmynafræðinni
- Bounded Contexts geta innihaldið fleiri Bounded Contexts
  - Eitt aggregate gæti mögulega verið sér context
  - Viljum byrja með stærri service og brjóta upp eftir þörfum

Bounded Contexts – Shared Models frh.

Innri virkni sem er ekki deilt á milli context-a

Innra reprsentaion af Stock Item

Stock Item verður *shared model* á milli context-a (hægt að hugsa um sem DTO)

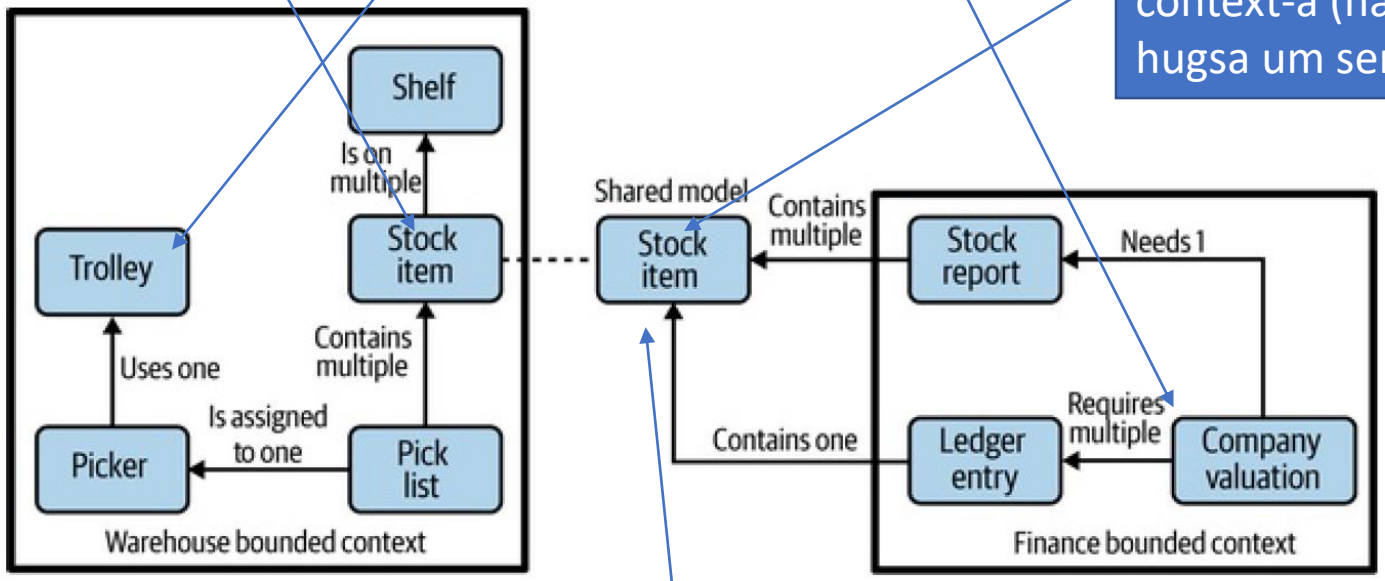


Figure 2-14. A shared model between the finance department and the warehouse

Warehouse birtir bara það sem þarf (representation af Stock Item)

Bounded Contexts – Shared Models frh.

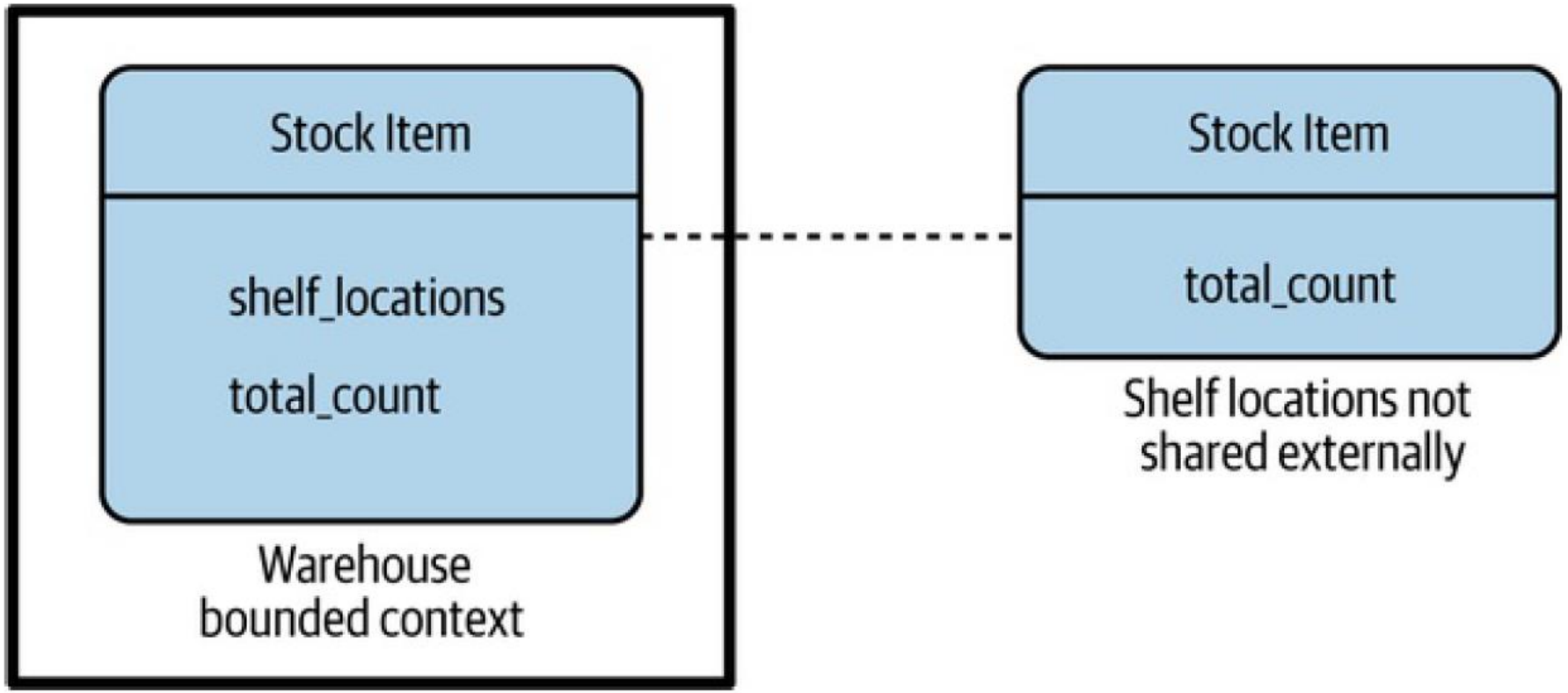


Figure 2-15. A model that is shared can decide to hide information that should not be shared externally

Bounded Context - Shared Models frh.

- Sama model / hugmynd getur birst í mörgum contexts
- T.d. Bæði *Finance* og *Warehouse* þurfa að vita um *Customer*
  - Hvorugt hefur þó umsjón yfir customer domain-inu
- Þessi model geta haft mismunandi merkingar
  - Gætu þ.a.l verið kallað mismunandi hluti
  - T.d. Customer í finance gæti verið kallað Recipient í Warehouse
  - Standa samt fyrir sama entity-ið
- Hvert context getur geymt upplýsingar um þetta entity
  - Upplýsingarnar geta verið öðruvísi
- Hægt að binda saman með *global customer*
  - Customer domain-ið í umsjá Customer Service

Warehouse Context inniheldur Inventory Context og Shipping Context

Getum falið uppbrotið með API layer

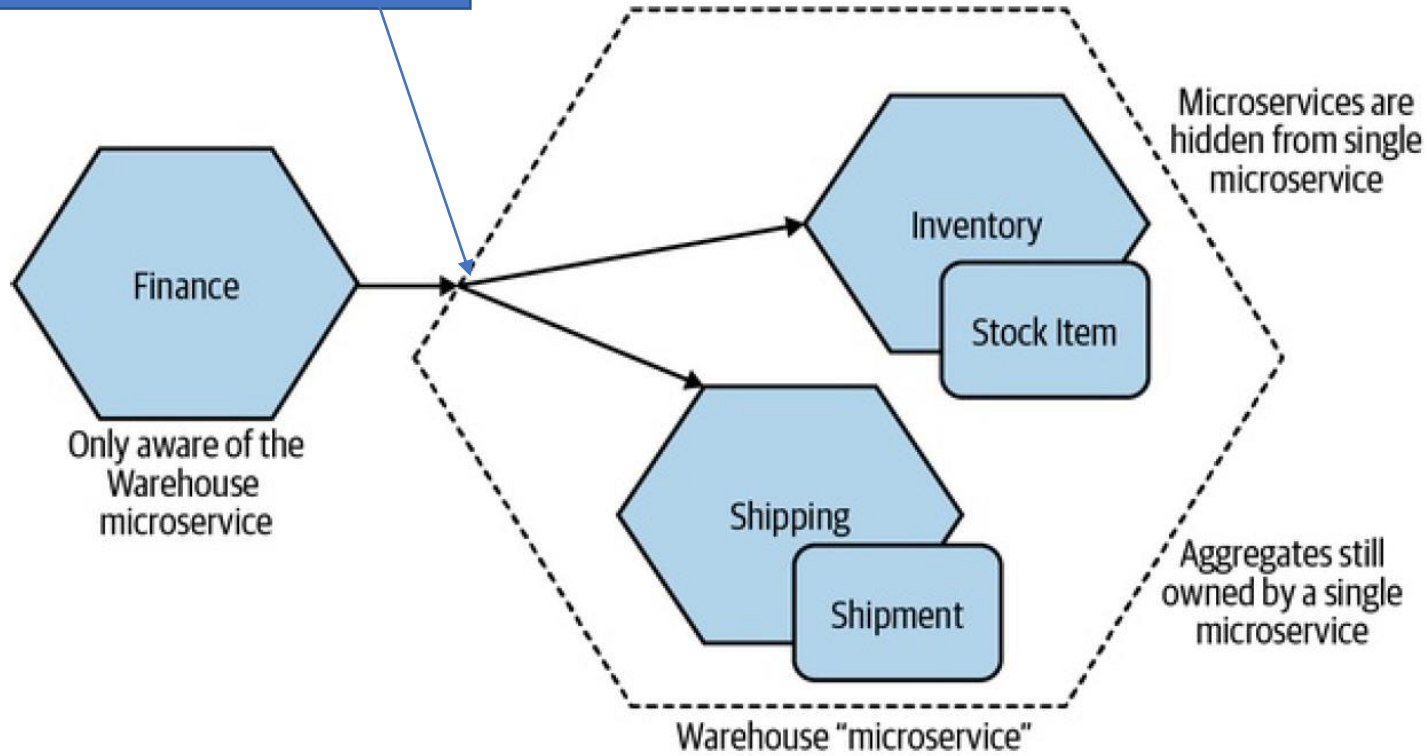


Figure 2-16. The `Warehouse` *service internally has been split into* `Inventory` *and* `Shipping` *microservices*

Event Storming

- Leið til að finna **domain model** og **bounded contexts**
- Allir enda með **sameiginlega sýn** og ***ubiquitous language***
- Góð leið til að model-a event driven kerfi
  - En virka líka fyrir request-response

Event Storming Ferli

- Allir saman í herbergi
  - forritarar, architects, product owners, stakeholders ...
- Fyrst finna domain ***events***
  - order placed, payment received...
- Næst finna ***commands*** sem valda event-um
  - Command er ákvörðun gerð af notanda kerfisins (manneskja)
  - Hér viljum við fá hugmyndir frá *key stakeholders*
- Næst finnum við ***aggregates***
  - Event-in og command-in byrja að mynda aggregate-in
  - T.d. ***Order*** *Placed*
  - Commands og events eru hópuð saman hjá aggregate-unum sínum
- Næst eru aggregates hópuð saman í bounded contexts

Gott að nota mismunandi liti fyrir events, commands og aggregates

Alternatives to Domain Boundaries

- Volatility
  - Viljum kannski draga út parta sem fara í gegnum margar breytingar
- Data & Security
  - Sum gögn eru viðkvæmari en önnur
  - Viljum kannski einangra viðkvæm gögn í einu service
  - Getum beint sérstakri öryggis athygli í þeim service
- Technology
  - Þurfum kannski sérstaka tækni fyrir ákveðna virkni
  - Technological Heteroginity kemur sterkt inn hér
- Organizational
  - Viljum ekki að mörg teymi eigi saman service
  - Getum model-að service-in okkar eftir organizational strúktúr
  - Eins gætum við breytt organizational strúktur til að fá service boundary-in sem við viljum

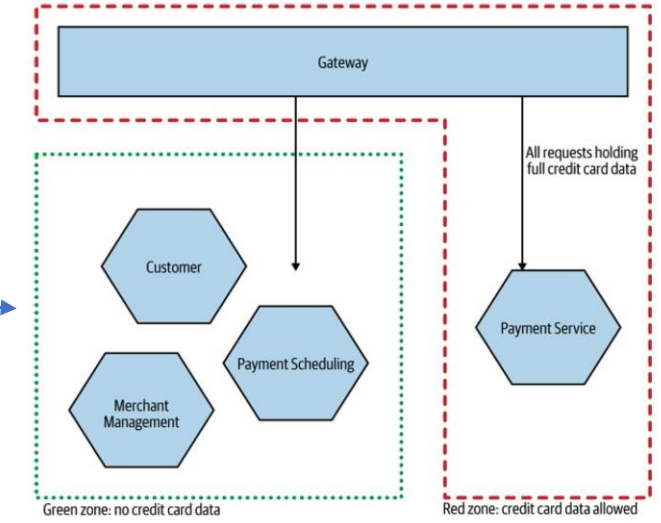


Figure 2-17. PaymentCo, which segregates processes based on its use of credit card information to limit the scope of PCI requirements