Microservices

Communication Styles

Hönnun og smíði hugbúnaðar

Haust 2022

Microservice Communication

- Mikilvægur þáttur í hönnun microservice-a
- Getur verið tricky
- Nokkrar leiðir til að velja um
  - REST
  - gRPC
  - GraphQL
  - Events (t.d. RabbitMQ, Kafka)

From In-Process to Inter-Process

- Ekki eins að kalla í gegnum netið og að kalla á *local* föll
- Sumt sem krefst endurhugsun við monolith sundurliðun
- Performance
  - *Network calls* taka lengri tíma
  - Þurfum að serialize-a / deserialize-a gögn
  - Getum ekki gert 1000 köll
  - *Latency is not zero*
- Data size
  - Gögn mega ekki vera of stór
  - Tekur tíma að senda í gegnum netið
  - Tekur tíma að serialize-a / deserialize-a

From In-Process to Inter-Process frh.

- Interface Breytingar
  - Interface breytingar erfiðari
  - Með in-process gátum við notað refactor tól
  - Með Inter-Process þurfum við að passa að brjóta ekki clients
  - Með Inter-Process gætum við þurft *step-lock deployment* eða versioning

- Error Handling
  - Erfiðara að meðhöndla villur
  - Villur geta nú verið út af network vandamálum
  - Villur geta nú átt sér stað ef eitt service er niðri

From In-Process to Inter-Process frh.

"A fallacy is something that is believed or assumed to be true but is not."

- Fallacy 1: The Network Is Reliable
- Fallacy 2: Latency is Zero
- Fallacy 3: Bandwith is infinite
- Fallacy 4: The network is secure
- Fallacy 5: The topology never changes
- Fallacy 6: There is only one administrator
- Fallacy 7: Transport cost is zero
- Fallacy 8: The network is homogeneous

Styles of Communication

- Synchronous
  - *blocking*
  - Eitt service kallar á annað **og bíður** eftir svari

- Asynchronous
  - *Non-blocking*
  - Eitt service kallar á annað og getur haldið áfram að vinna þangað til svar berst
  - Svar þarf ekki að berast
  - Þegar / ef svar berst getur service-ið unnið úr því þá

Synchronous

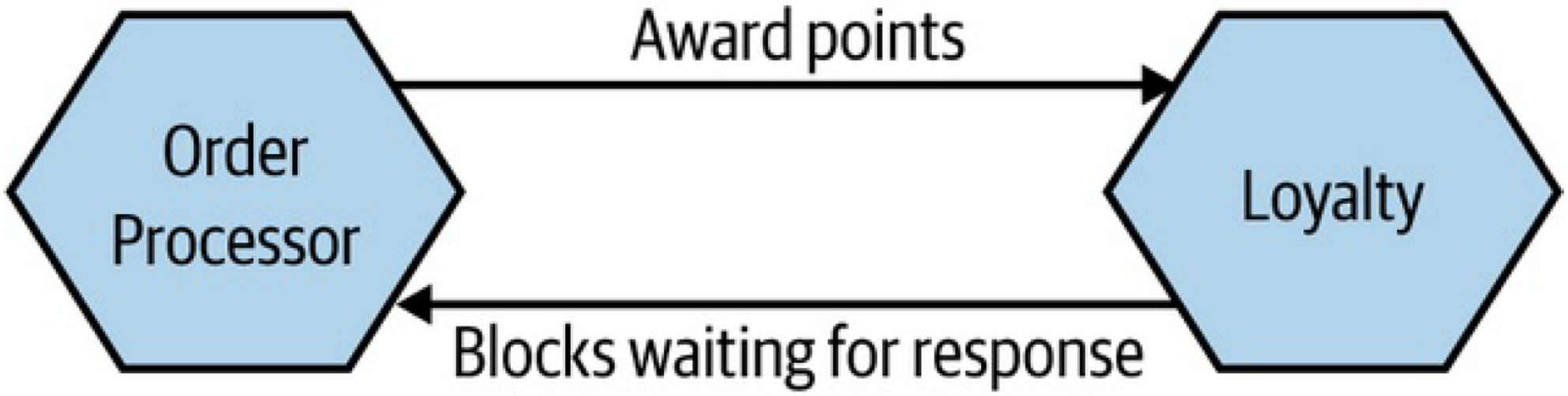


Figure 4-2. `Order Processor` *sends a synchronous call to the* `Loyalty` *microservice, blocks, and waits for a response*

Synchronous Kostir og Gallar

- Kostir
  - Þurfum oft svar til að halda áfram
  - Viljum oft tryggja að *request* gekk upp
  - Einfalt (línuleg hugsun)

- Gallar
  - Service-ið sem er kallað á þarf að vera uppi (*temporal coupling*)
  - Service-ið sem kallaði þarf að vera uppi þegar svarið berst
  - Getur verið hægt
  - Einn veikur hlekkur brýtur allt
  - Einn hægur hlekkur hægir á öllu
  - Eitt Architectural Quantum
  - Hætta á *resource contention*

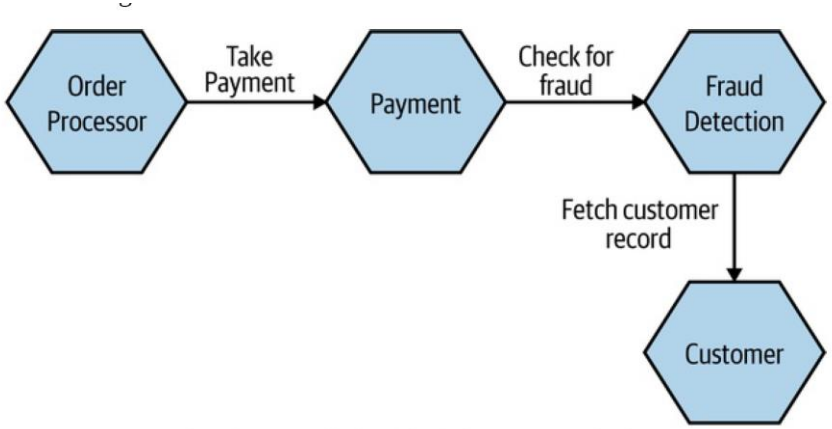


Figure 4-3. Checking for potentially fraudulent behavior as part of order processing flow

Asynchronous

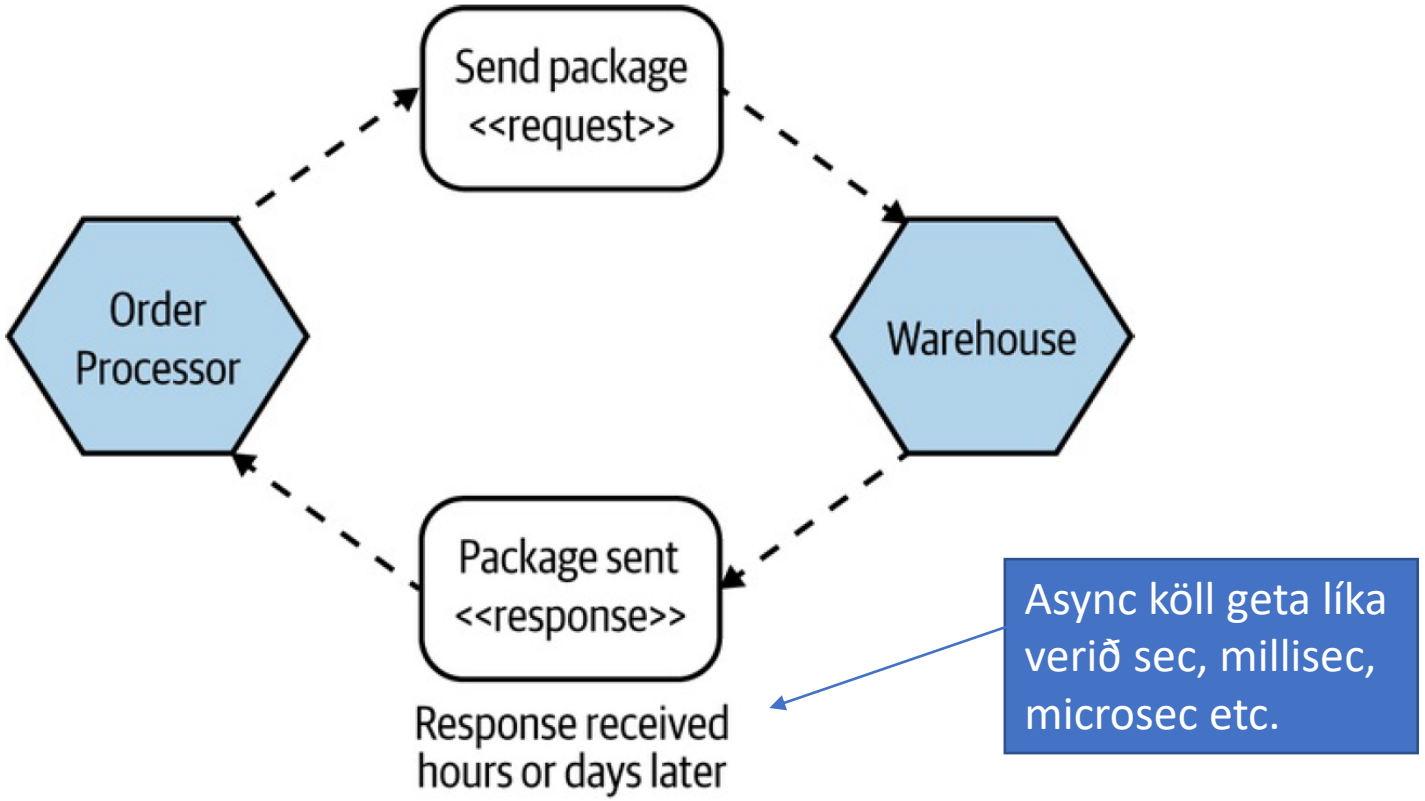


Figure 4-5. The `Order Processor` kicks off the process to package and ship an order, which is done in an asynchronous fashion

Asynchronous kostir og gallar

- Kostir
  - Getur komist framhjá temporal coupling
  - Minna resource contention (hægt að vinna í öðrum hlutum á meðan)
  - Hraðara
  - Getur komist framhjá veikum hlekkjum
  - Hægt að hafa mörg architectural quantum
  - Gerir mögulegt að útfæra *long running processes*
- Gallar
  - Flóknara
  - Krefst annan hugsunarhátt
  - Getur gert kerfið ólæsilegra

Styles of Communication frh.

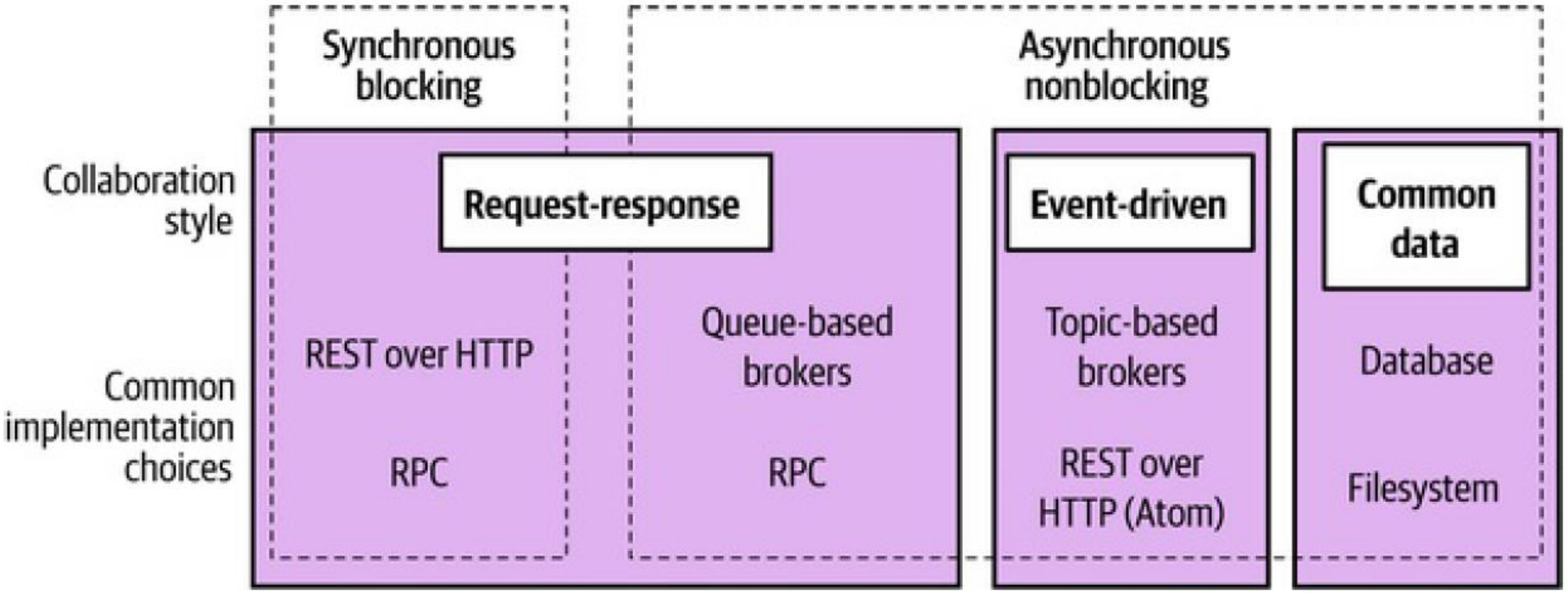


Figure 4-1. Different styles of inter-microservice communication along with example implementing technologies

Styles of Communication frh.

- Common Data
  - Service tala saman í gegnum deild gögn
  - T.d. deildur gagnagrunnur
  - Fer á móti microservice hugmyndafræði
  - Er asynchronous
- Request-Response
  - Service sendir fyrirspurn (e. *request*) á annað service **og fær** svar (e. *response*)
  - Getur verið synchronous eða asynchronous
- Event-Driven
  - Service *publish-ar* event-i sem önnur service hlusta eftir
  - Publisher fær ekki neitt svar
  - Publisher veit ekki einu sinni af subscribers
  - Er asynchronous (fire and forget)
  - Að koma event-um á topic / queue-ið er þó yfirleitt synchronous

Common Data

Fer á móti microservice hugmyndafræðinni

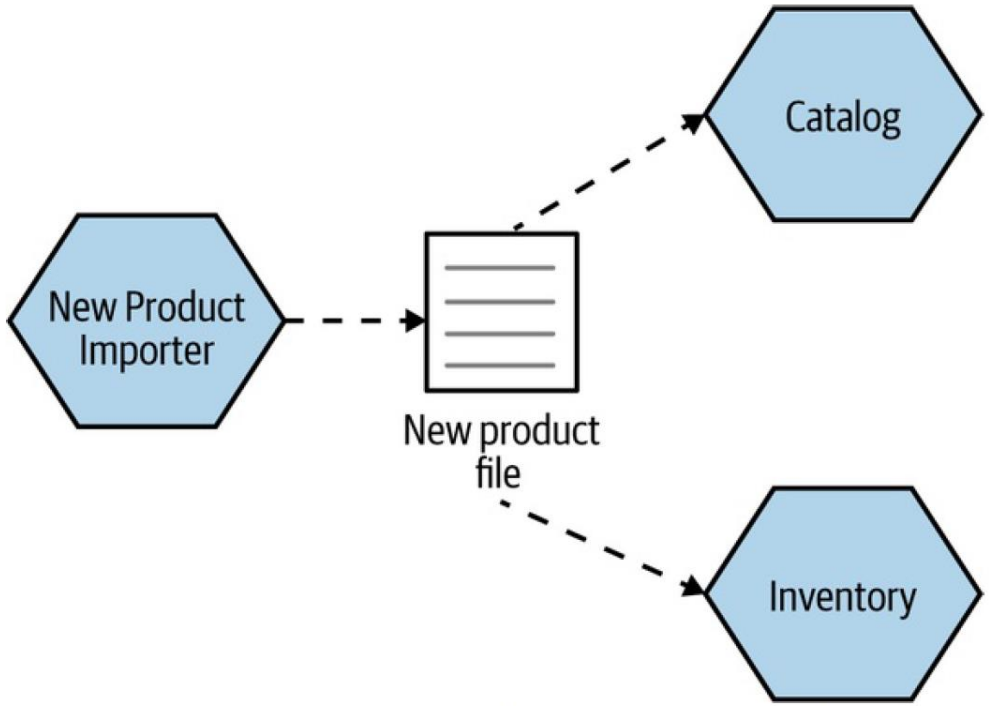


Figure 4-6. One microservice writes out a file that other microservices make use of

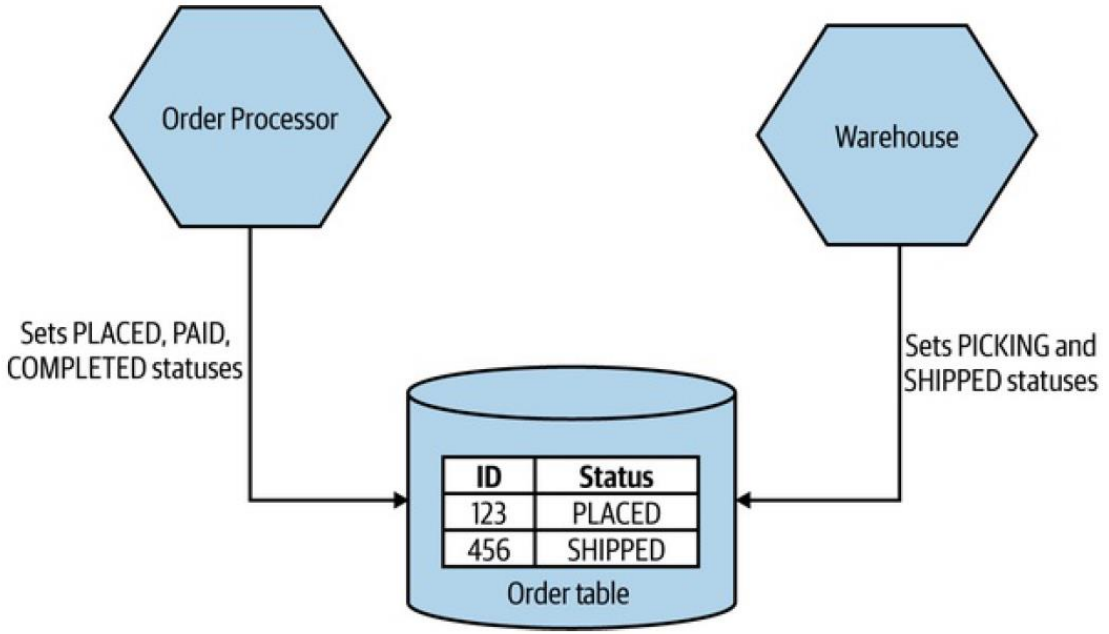


Figure 4-7. An example of common coupling in which both `Order Processor` *and* `Warehouse` *are updating the same order record*

Common Data kostir og gallar

Kostir

- Mjög einfalt
- Auðvelt að deila miklu magni af gögnum (líka hægt að leysa með Kafka)
- Interoperability (gömul kerfi geta líka notað þetta)

Gallar

- **Coupling**
- Ekki best fyrir *low latency*
- Engin frábær leið til að vita að ný gögn eru í boði
  - CRON job
  - Poll-a
  - Senda request um að gögn eru í boði

Request Response

Getur bæði verið synchronous og asynchronous

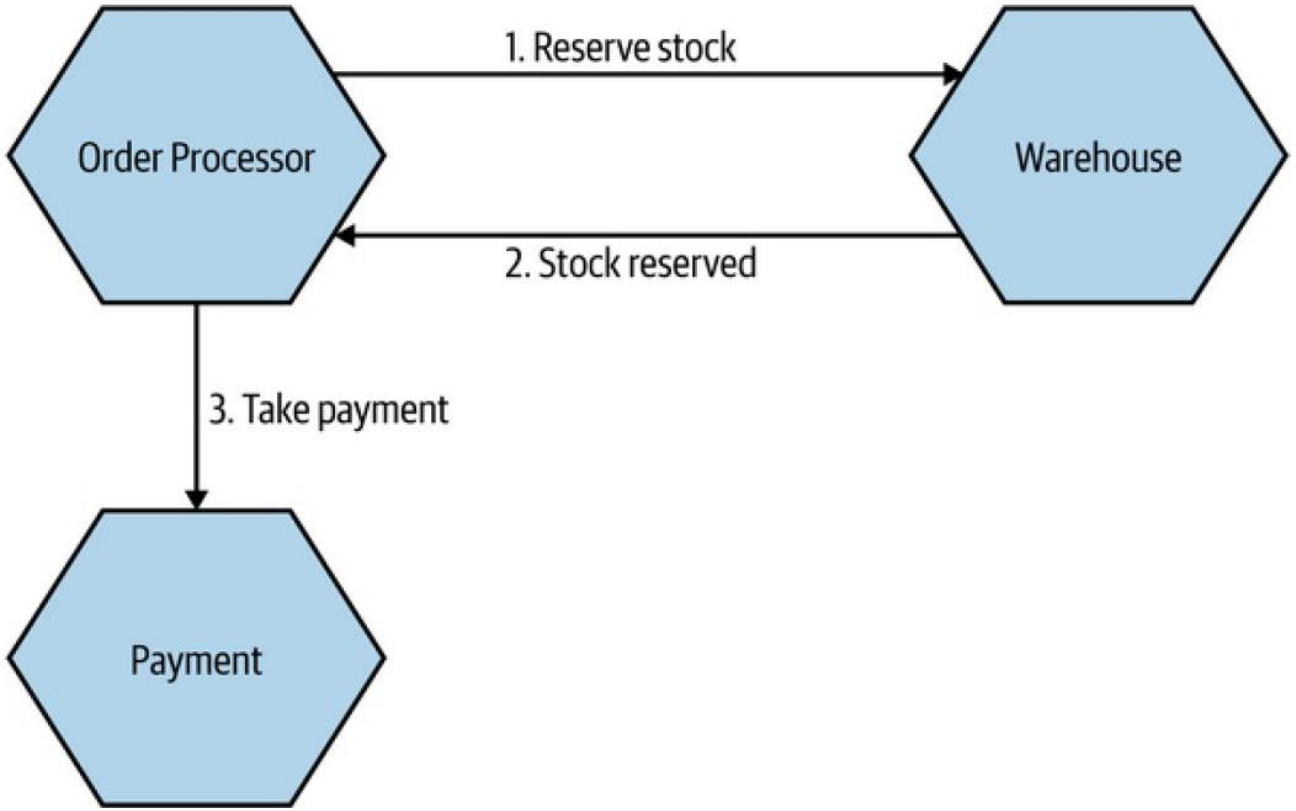


Figure 4-9. `Order Processor` *needs to ensure stock can be reserved before payment can be taken*

Request Response event útfærsla

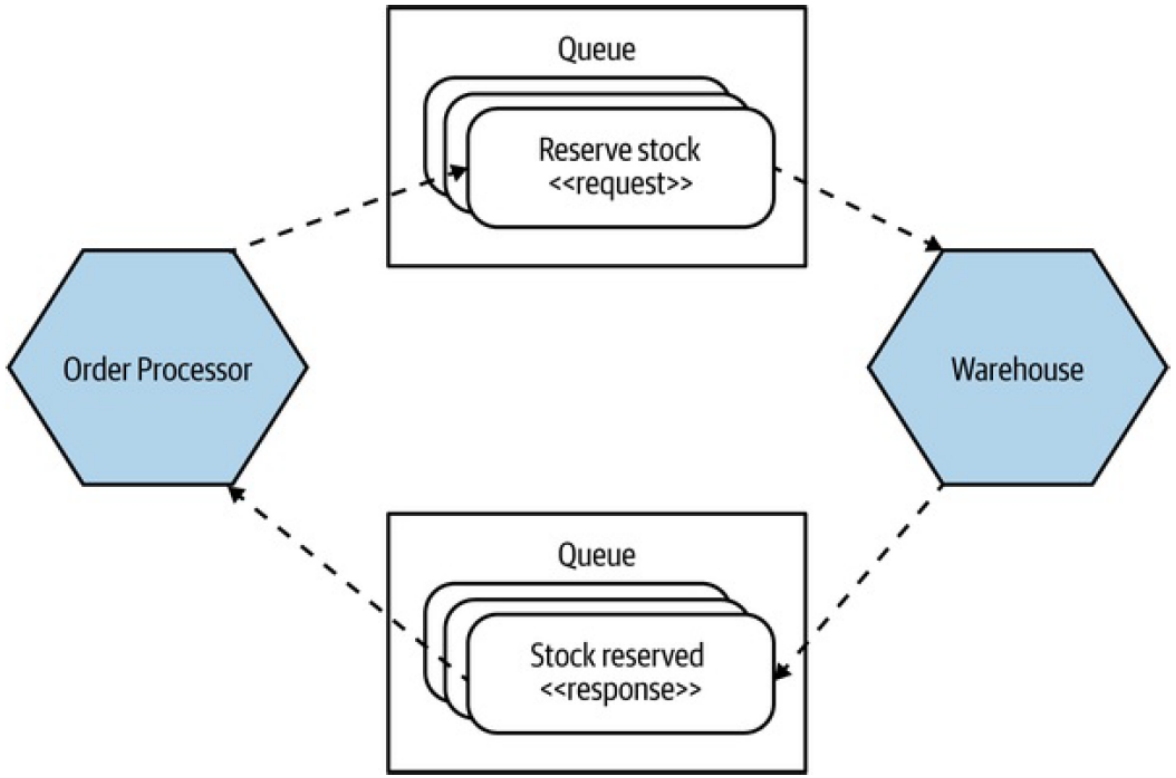


Figure 4-10. Using queues to send stock reservation requests

Request Response kostir og gallar

- Kostir
  - Þurfum oft svar til að halda áfram
  - Viljum oft tryggja að *request* gekk upp
  - Einfalt
  - Minni coupling en common data

- Gallar
  - Meiri coupling en event driven
  - Fylgir oft temporal coupling (hægt að leysa með events)

Event Driven

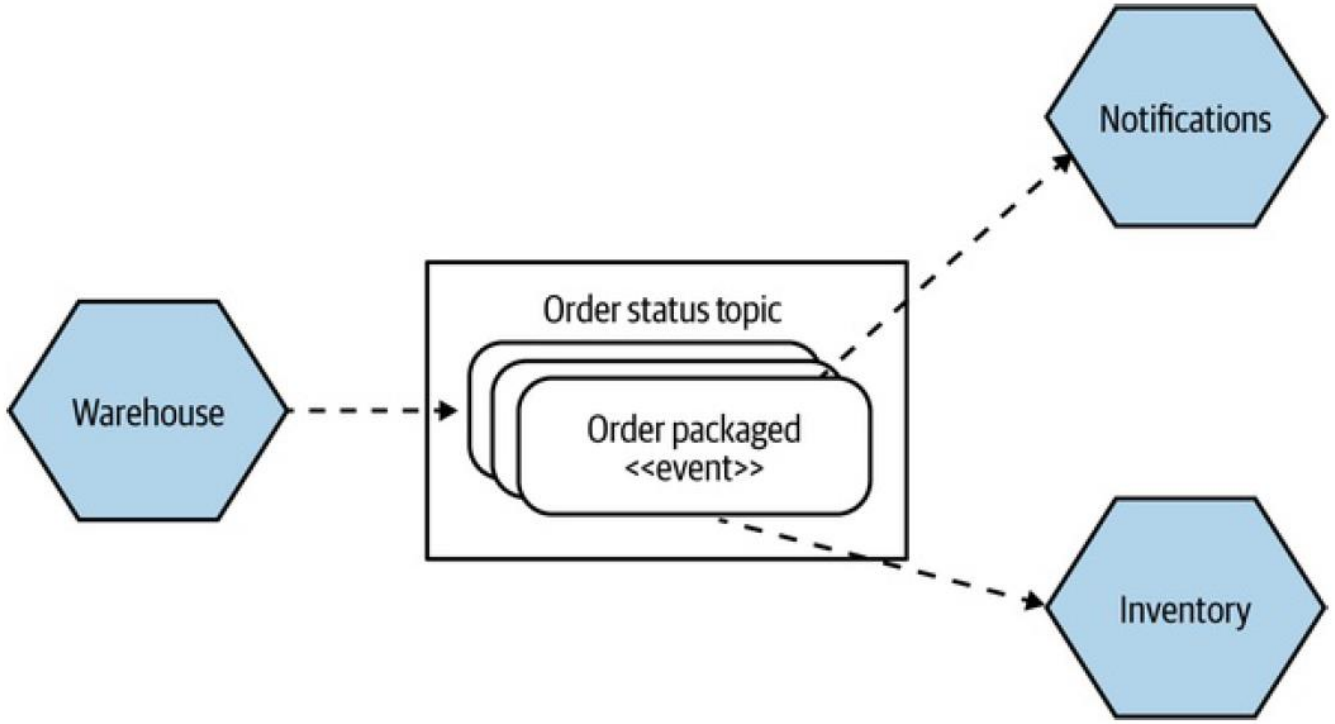


Figure 4-11. The `Warehouse` *emits events that some downstream microservices subscribe to*

Event Driven kostir og gallar

- Kostir
  - Eins decoupled og mögulegt
  - Hægt að replay-a events
  - Ekkert temporal coupling
  - Mjög hátt throughput

- Gallar
  - Flókið
  - Meira overhead
  - Erfiðara að testa

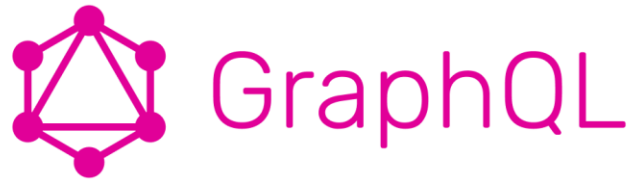
Styles of Communication frh.

- Mjög eðlilegt að nota marga style-a
  - Þurfum ekki að velja að nota bara async
  - Þurfum ekki að velja að nota bara event driven
  - Etc.

- Notum það sem hentar best hverju sinni

- **Mix and Match**

Request Response tækni



REST

- **RE**presentational **S**tate **T**ransfer
- Vinsælasta communication method
- Yfirleitt notað yfir HTTP
  - GET, POST, PUT, DELETE
- Allt hugsað sem aðgerðir á *resources*
  - GET api/orders
  - POST api/orders
  - GET api/orders/{id}/customer
  - …

REST frh

- Representation of State
  - Það sem er skilað er *representation* af undirliggjandi gögnum
  - Stuðlar að decoupled design
  - Stuðlar að information hiding
  - **Á í raun við um alla aðra communication still**

- JSON
  - Sent sem json gögn
  - Þurfum að serialize-a og deserialize-a

- HATEOAS
  - **H**ypermedia **a**s **t**he **e**ngine **o**f **a**pplication **s**tate
  - Principle sem segir að gögn ættu að innihalda link-a á önnur gögn
  - Hugmyndin er að auka decouplingu á milli server og client
  - T.d. order.customer = https://something.com/api/order/123/customer
  - Þetta principle er oft ekki útfært

REST dæmi

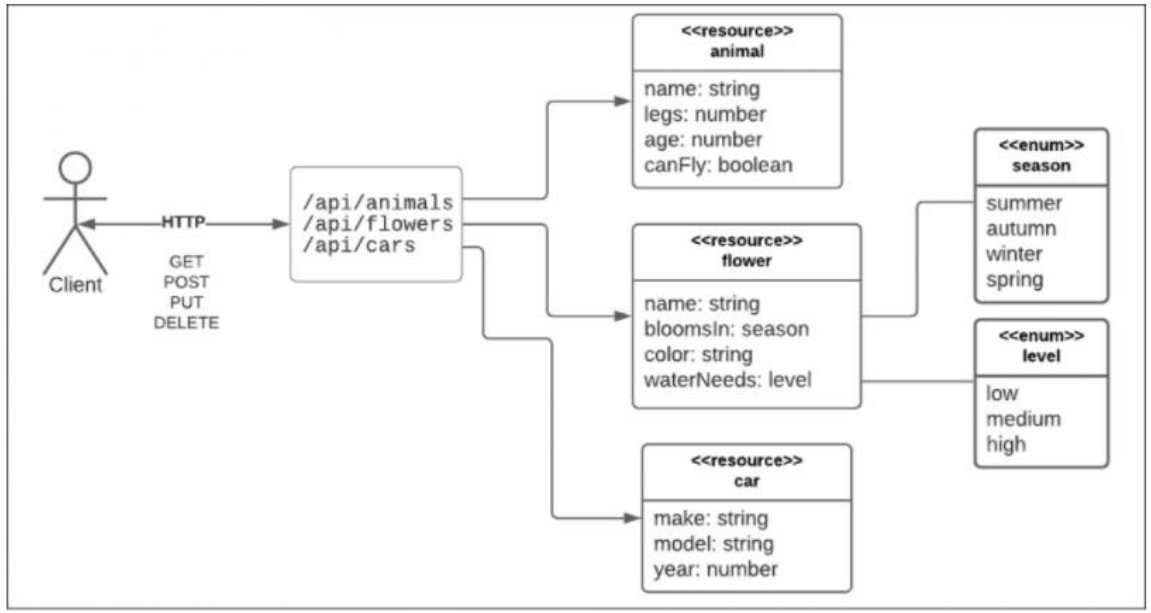


Figure 1: REST uses standard HTTP methods to work with server-side resources

```
{
  "car": {
    "vin": "KNDJT2A23A7703818",
    "make": "kia",
    "model": "soul",
    "year": 2010,
    "links": {
      "service": "/cars/KNDJT2A23A7703818/service",
      "sell": "/cars/KNDJT2A23A7703818/sell",
      "clean": "/cars/KNDJT2A23A7703818/sell"
    }
  }
}
```

REST stutt python dæmi

server.py

```python
app = FastAPI()


class Customer(BaseModel):
    id: Optional[int]
    name: str
    ssn: str
    phone_number: str


customers = []


@app.get("/customers/{id}")
def get_best_number(id: int):
    return next((customer for customer in customers if customer.id == id), None)


@app.post("/customers")
def save_best_number(body: Customer):
    body.id = __get_next_id()
    customers.append(body)
    return body.id


def __get_next_id():
    return 0 if len(customers) == 0 else max([customer.id for customer in customers]) + 1


if __name__ == '__main__':
    uvicorn.run('server:app', host='0.0.0.0', port=8000, reload=True)
```

client.py

```python
import requests

create_customer_response = requests.post('http://localhost:8000/customers', json={
    'name': 'test_name',
    'ssn': '0000000000',
    'phone_number': '1111111'
})

id = int(create_customer_response.text)


get_customer_response = requests.get(f'http://localhost:8000/customers/{id}')
print(get_customer_response.json())
```

REST kostir og gallar

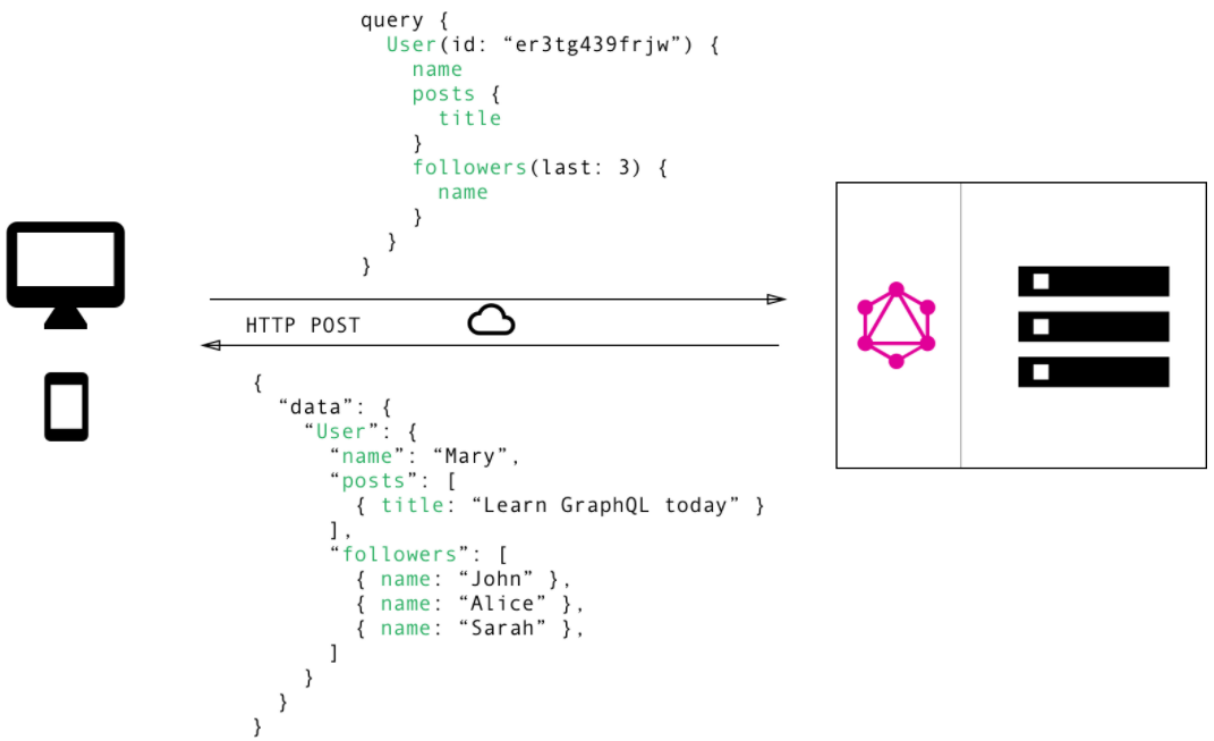
- Kostir
  - Einfalt
  - Lang vinsælast
  - Interoperability

- Gallar
  - Hægara en gRPC (því text based)
  - Getur ekki sótt nákvæmlega það sem þú þarft
  - Ekki gott support fyrir code generation

GraphQL

- GraphQL er query language fyrir APIs
- Hægt að hugsa um sem SQL fyrir network calls
- Leyfir okkur að sækja nákvæmlega þau gögn sem við þurfum
- Lendum ekki í under-fetching eða over-fetching vandamálum
- Notar HTTP
- Alternative við GraphQL er Backend for Frontend (BFF)

GraphQL dæmi



GraphQL kostir og gallar

- Kostir
  - Hver client sækir nákvæmlega það sem hann þarf
  - Breytingar í framenda/client auðveldar
  - Hátt flexibility og adaptability
  - Insightful analytics í data notkun
  - Margir client-ar geta notað sama bakenda

- Gallar
  - Flóknara en REST
  - Ekki jafn mikið support og REST (orðið mjög gott samt)
  - Hægara en gRPC (því text-based)
  - Getur verið minna performant (gæti þurft mörg db köll)
  - Caching og CDNs erfiðara
  - Getur leitt til database wrappers

gRPC

- ***R**emote **P**rocedure **C**alls*
- Lætur remote calls líta út eins og local calls
- Vinsælasta RPC lausnin í dag
- Hægt að generate-a clients
- Mjög performant
- Til fleiri RPC útfærslur (t.d. SOAP, Java RPC)

gRPC stutt python dæmi

examples/protos/helloworld.proto

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

greeter_server.py .

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

  def SayHello(self, request, context):
    return helloworld_pb2.HelloReply(message='Hello, %s!' % request.name)

  def SayHelloAgain(self, request, context):
    return helloworld_pb2.HelloReply(message='Hello again, %s!' % request.name)
...
```

greeter_client.py

```
def run():
  channel = grpc.insecure_channel('localhost:50051')
  stub = helloworld_pb2_grpc.GreeterStub(channel)
  response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'))
  print("Greeter client received: " + response.message)
  response = stub.SayHelloAgain(helloworld_pb2.HelloRequest(name='you'))
  print("Greeter client received: " + response.message)
```

https://grpc.io/docs/languages/python/quickstart/

gRPC kostir og gallar

- Kostir
  - Mjög performant
  - Code generation support

- Gallar
  - Flóknara í uppsetningu
  - Hætta á að halda að maður er að gera local kall
  - Ekki öll tækni styður gRPC (t.d. Legacy dót)
  - Ekki hægt að gera gRPC köll í framenda (í browser)

Tölum um event driven tækni seinna