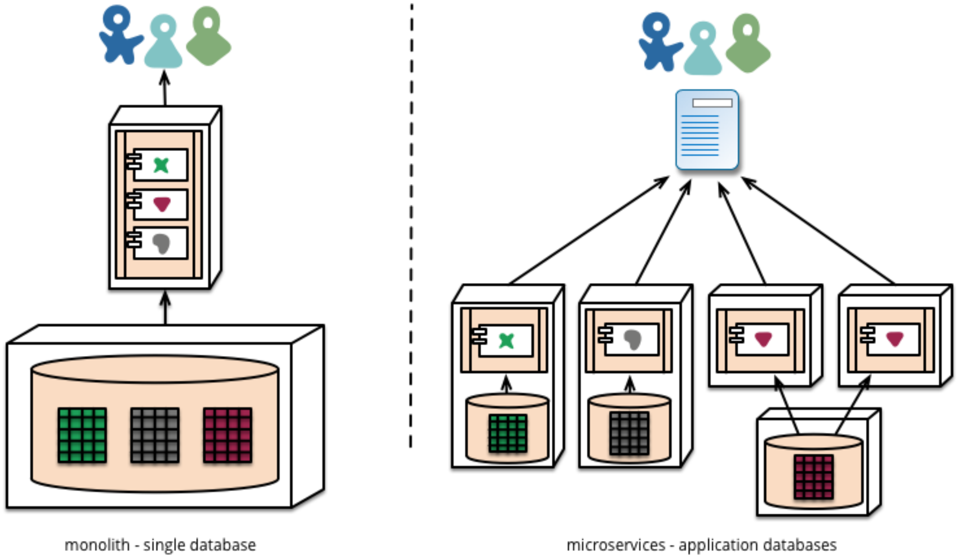
Microservices

Frank Walsh

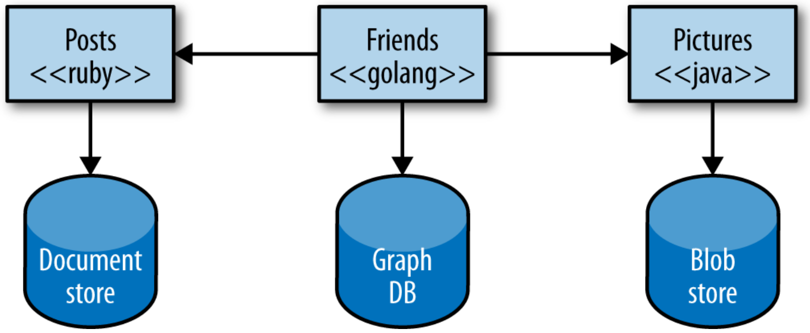
What's a Microservice

- Do one thing and one thing well
- Defined service boundaries
  - Usually based on business boundaries
- NEVER strays outside well defined boundary
  - e.g. contact service does contact stuff
  - No function creep
  - Small enough but no smaller..
  - Small enough for the codebase to be manageable
  - Not so small that you have too many moving parts
- Autonomous
  - A separate entity, isolated from other services.
  - Can change independent of other services

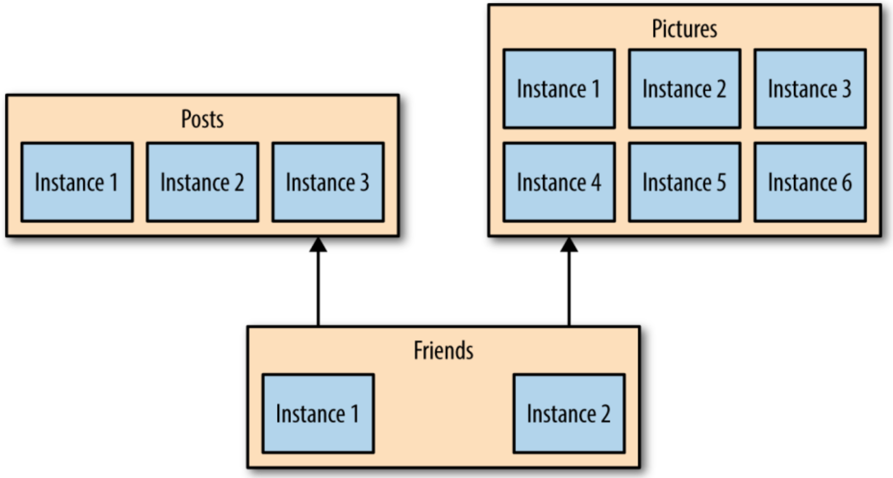

monolith - single database
microservices - application databases

Why Microservices

- Technology Heterogeneity



- Resilience
  - Failure should not cascade through the system
  - Service boundaries can act as a "bulkhead"
- Scaling
  - Can scale individual services if required, independent of other services in the system.



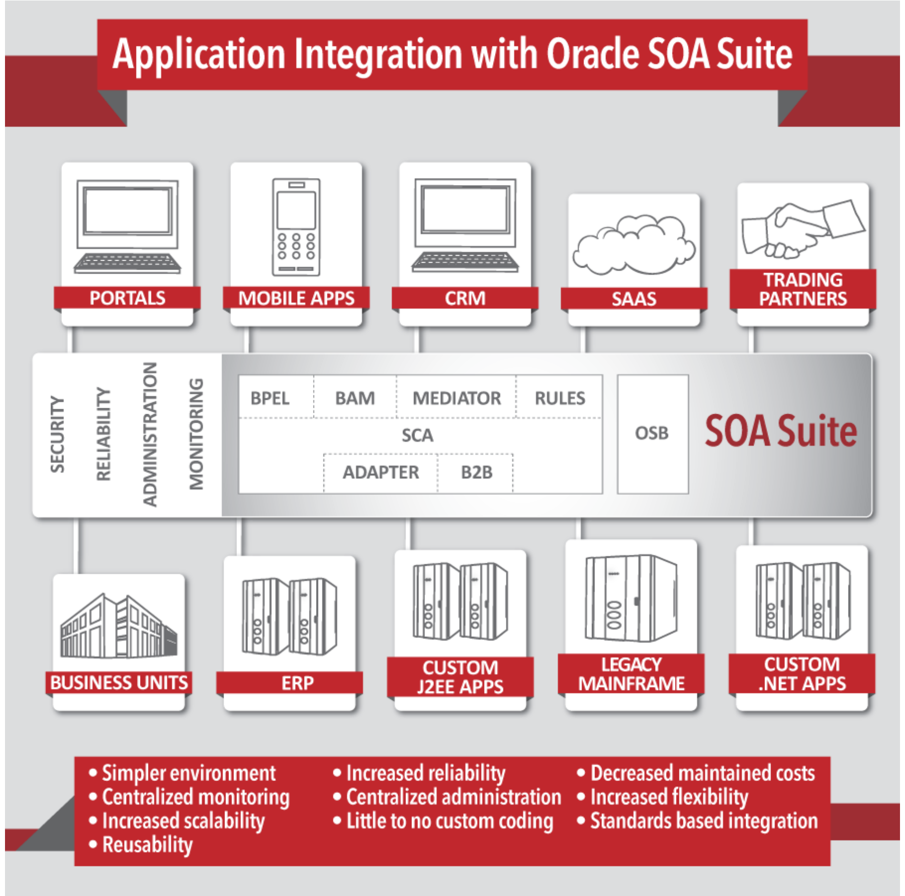
- Simplified Deployment
  - Changes do not require re-deployment of the whole system. Just the changed service(s).

Why Microservices 2

- Aligns with organisational structure
  - Small teams focused on smaller codebase alligned to come business function (e.g. contacts)
- Composable
  - Fine-grained services can be composed into many functions/uses without change.
  - Older, "course grained" services typlicallyt useful just in original purpose.
- Easy to replace
  - Because they're small, replacing a service has less risk/overhead.

Service Orientated Architecture (SOA)

- SOA is a mature architectural approach
- Multiple services collaborate to provide overall system
  - Transparent to user - User percieves just one application
- In principle, a good idea
  - Promotes reusability, composability
- Loss of popularity attributed to:
  - Vendor-driven products (
  - Lack of guidance on boundaries
  - Not enough granularity
- Overly complex stacks/products that required too much expensive tooling



Other Contenders

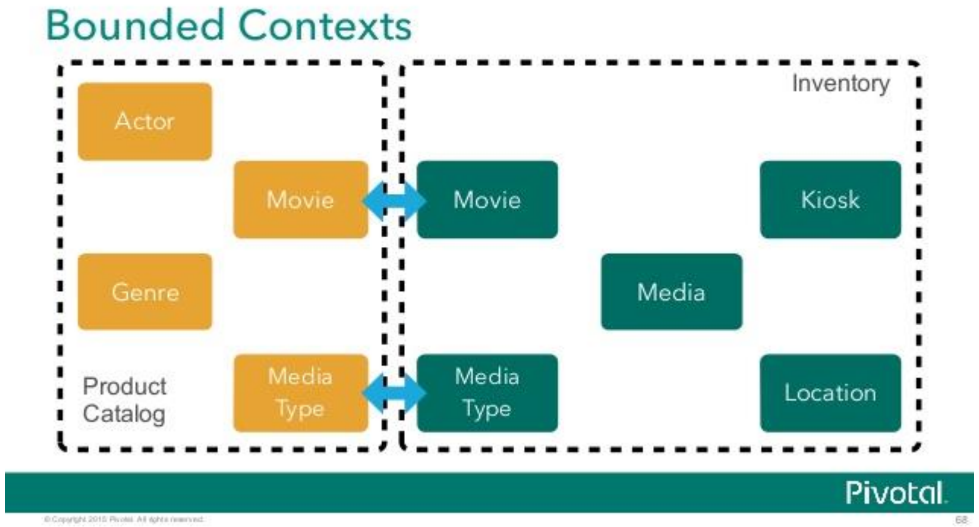
- Shared Libraries
  - Common, reusable code shared as libraries (e.g. the java archive, jar file)
  - Will always be popular and highly effective for some applications (e.g. javascript libraries)
  - Not heterogeneous – libraries are written in one language. Cant use a java library if I run with python.#
- Modules
  - You're already using them via Node Packet Manager
  - Will become tightly coupled with your code.

Service Modeling

- A good microservice should exhibit:
  - Loose coupling
    - Changes to one service should not require changes to another
  - High Cohesion
    - Related behaviour should be together
    - A change to behaviour (e.g. billing process) should be made in one place and NOT in lots of places.
- To do this you need to define boundaries

Bounded Context

- Any domain can be broken up into multiple bounded contexts
  - Each context contains
    - Things that are internal to that context(do not require communication with other contexts)
    - Things that are shared wth other contexts
- Each context has an explicit, shared boundary
  - You communicate with the context though the boundary.
- Analogous to biological cells
  - Communication pathways connected via membrane receptors.



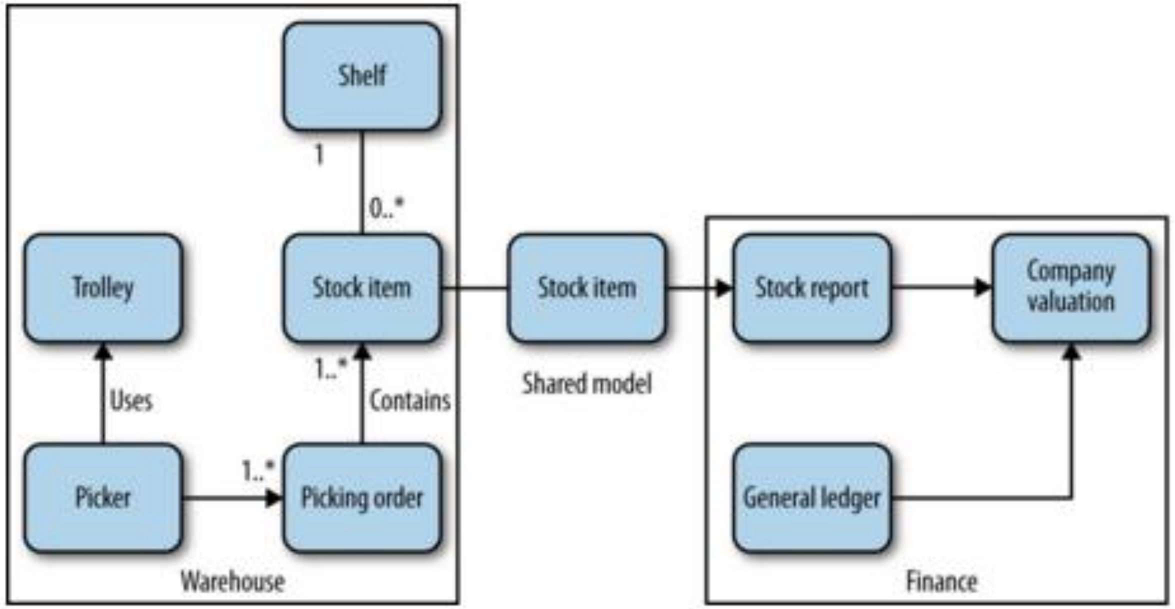
Bounded Contexts

Example:

- MusicCorp:
  - 2 separate contexts: Warehouse and Finance
- Warehouse functions:
  - Manage order shipping and returns
  - Receive goods
- Finance functions:
  - Payroll
  - Accounts
  - Reporting

External and internal concepts

- Internal concepts only need visibility inside a context.
- External concepts need visibility outside context:
  - Must be provided through an interface
- Example
  - Finance need to stock levels in warehouse for company valuations
  - Stock Items exposed through a *Shared Model*



External models and Code Modules

- Defining external models helps to:
  - Promote loose coupling
  - Identify boundaries
  - Promote cohesion(where similar, supporting things live)

Think function rather than data

- When a bounded context is defined, think of the capabilities it should provide:
  - e.g. warehouse should provide capability to get stock level; finance provides capability to get end-of-month accounts
- These capabilities may require interchange of data – a shared model.

Course Grained and Fine Grained Contexts

- Initially, easier to define fewer course-grained contexts that contain smaller, finer grained contexts
- It may be required to make them high level contexts...

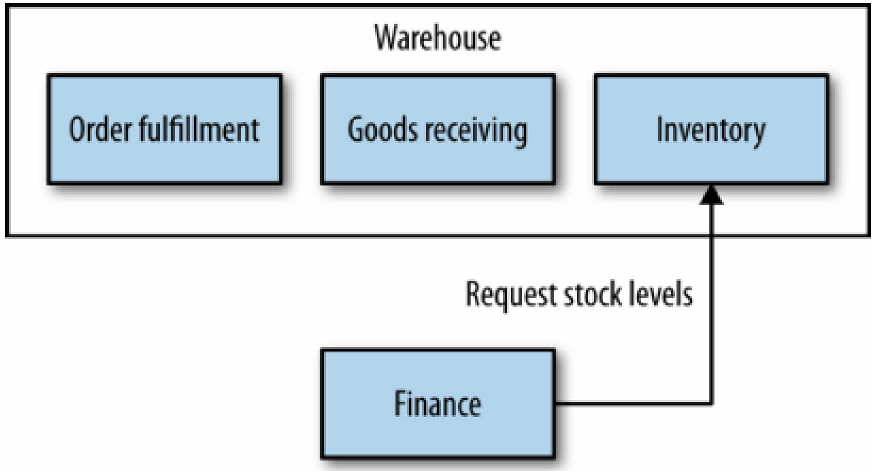
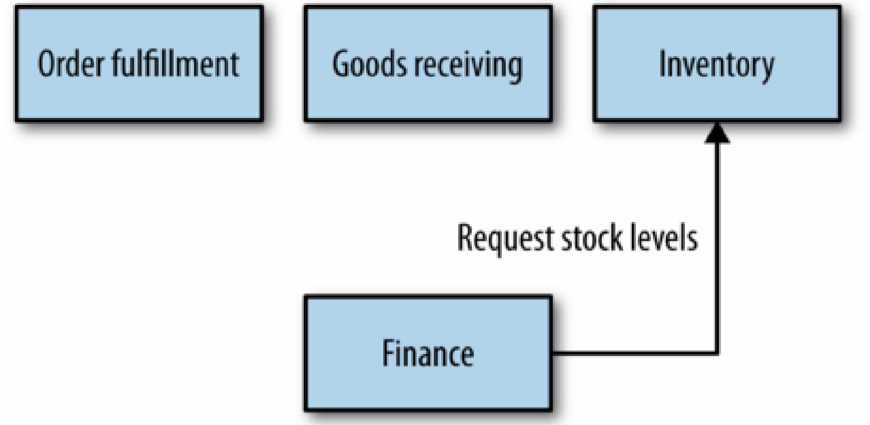


Figure 3-2. Microservices representing nested bounded contexts hidden inside the warehouse

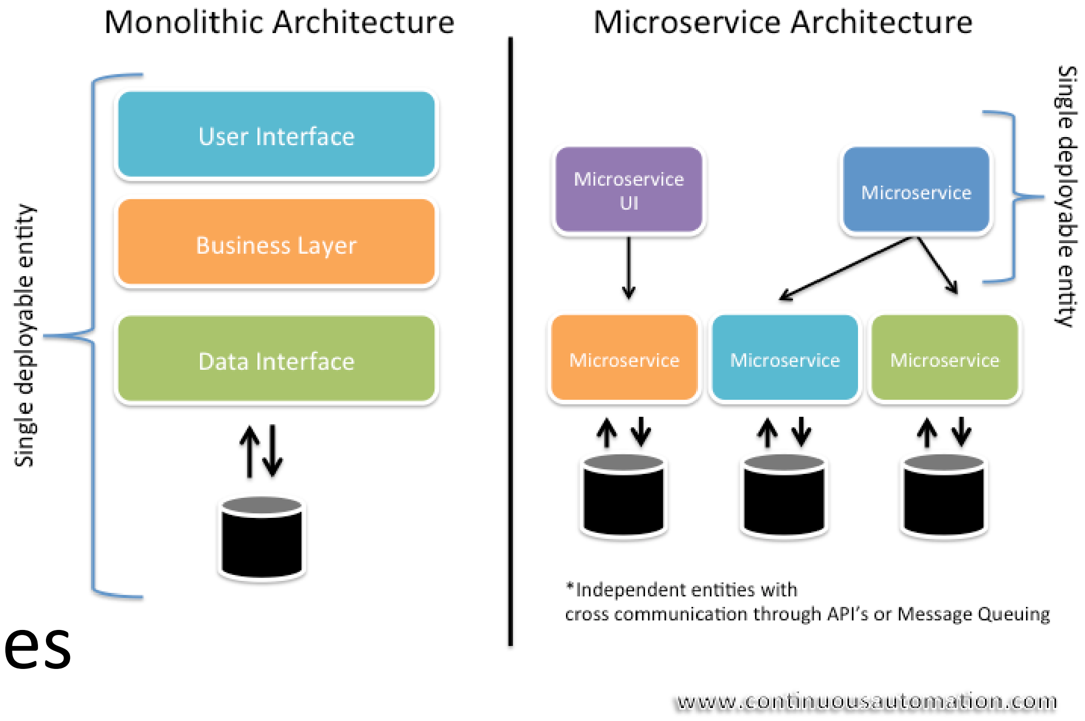


Problem with layered architectures

- Traditionally architectures(and teams) split in layered manner:
  - Front end web developers
  - Back end service/middleware developers
  - Database admins
- Can become tightly coupled, overly complex and brittle

PICTURES

Some times known as onion architectures (lots of layers and causes tears when cut though.

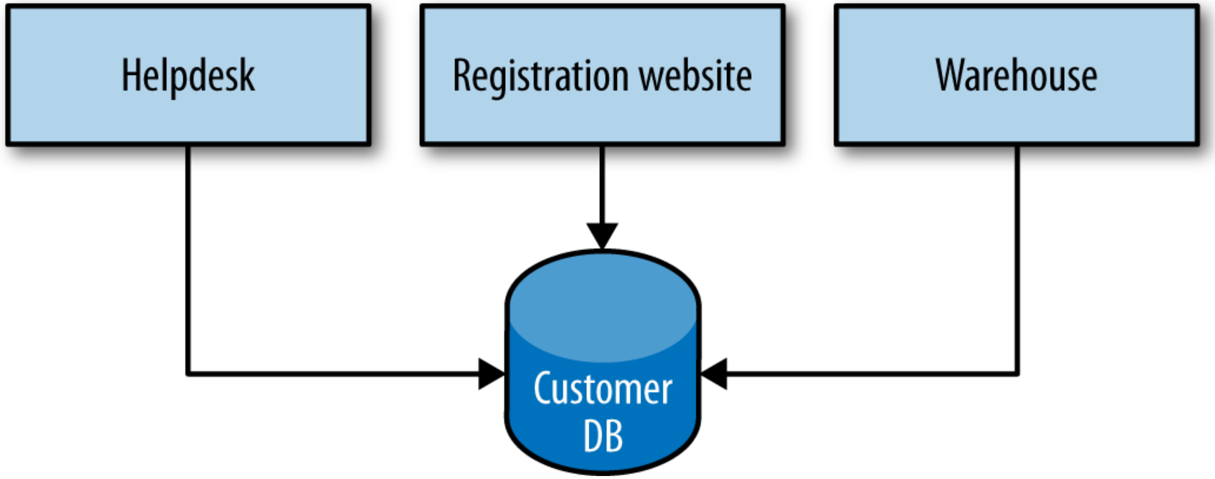


Integration

- Integrating services is critical to create distributed systems.
- We're looking at Node, but what's the best tools and tech. to build/integrate microservices.
- Whatever you choose, the following characteristics:
  - Avoid breaking changes
  - APIs should be technology agnostic
  - Simple APIs
  - Hide implementation details

The Shared Database

- Commonly used integration approach
- One database/one source of data
- All services reach into the DB for data



Shared Database

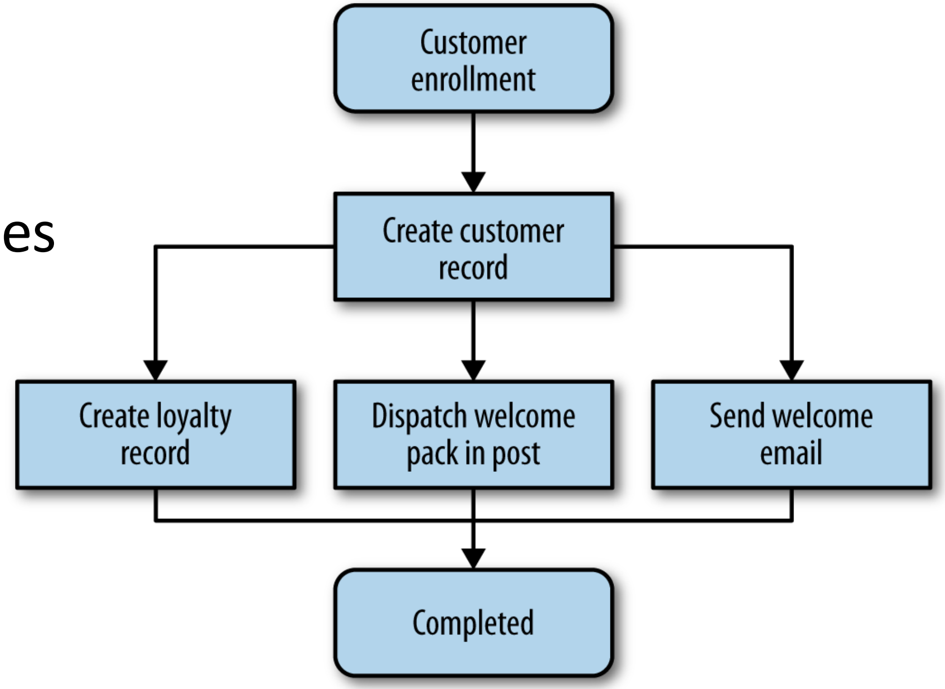
- Easy to implement. Simple in principle
- Issues
  - external parties to view and bind to internal implementation details
  - tied to a specific technology choice
  - business logic needs to be replicated in each client.
- Easy to share data but not so easy to share behaviour (e.g. rules about creating a customer/deleting a customer)

Synchronous vs. Asynchronous

- Should communication be synchronous or asynchronous?
  - synchronous communication, a call is made to a remote server, which blocks until the operation completes.
  - asynchronous communication, the caller doesn’t wait for the operation to complete before returning, and may not even care whether or not the operation completes at all.
- Usually based on two styles of collaboration
  - Request/response: A client initiates a request and waits for the response (usually synchronous but can be asynscronous)
  - event-based: client says this "event" happened and expects other parties to know what to do

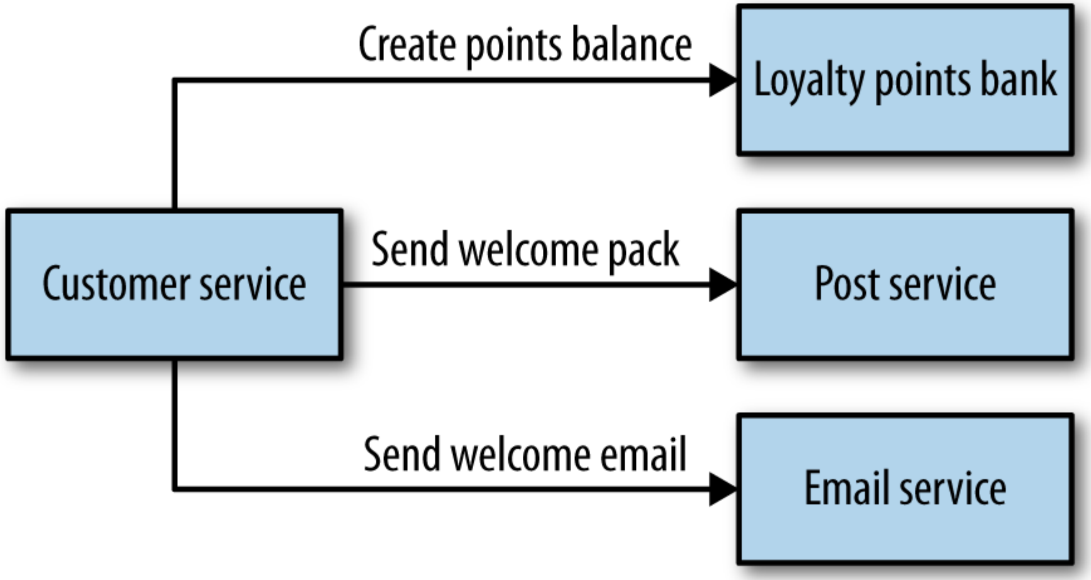
Orchestration Vs Choreography

- Creating a customer may involve:
  - A new record is created in the loyalty points bank for the customer
  - Postal system sends out a welcome pack
  - Send a welcome email to the customer
- Implementing this can be via
  Orchestration: central brain controls and drives the process
  Choreography: Inform each part of its job let it work out the details.



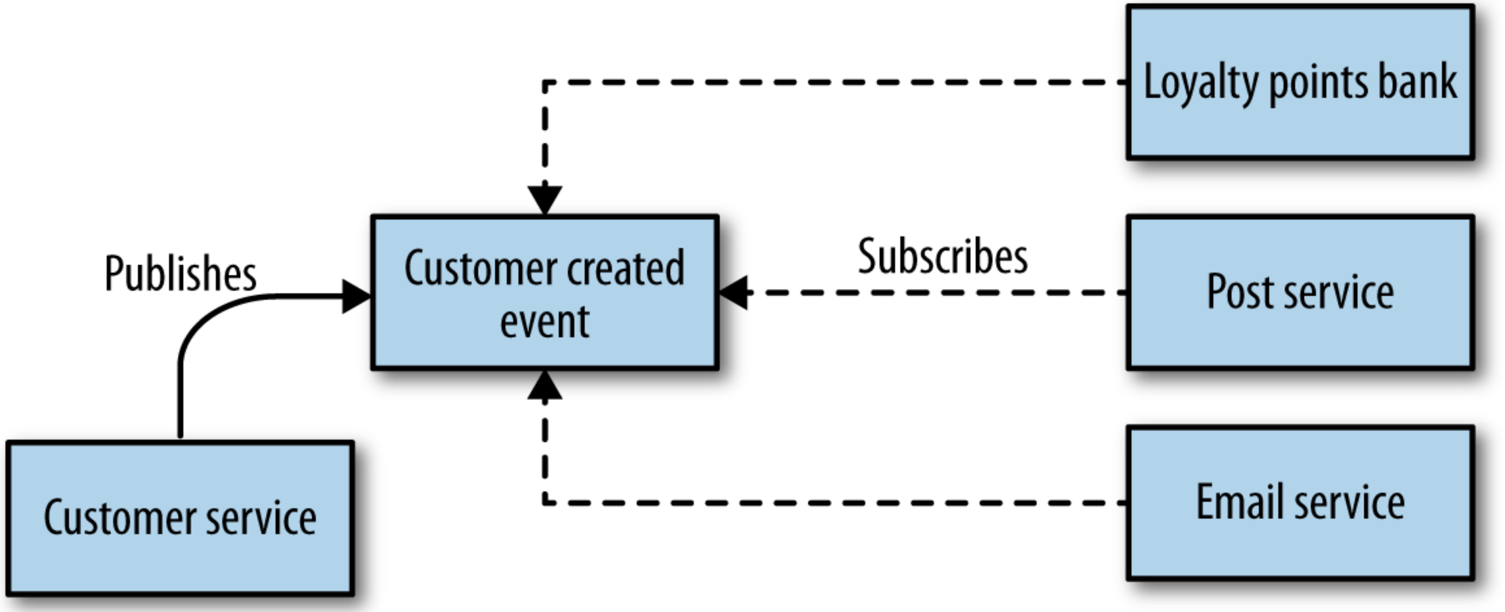
Orchestration

- Customer creation via. orchestration



Choreography

- Customer Creation via. Choreography



References

Microservices (nginx): https://www.nginx.com/blog/introduction-to-microservices/

Building Microservices: https://www.nginx.com/wp-content/uploads/2015/01/Building_Microservices_Nginx.pdf