Service-Oriented Architecture

Microservices

Lecture 06

Henry Novianus Palit

hnpalit@petra.ac.id

What are Microservices? *(1)*

- Microservices are small, autonomous services that work together
- Small (and focused on doing one thing well)
  - Despite a drive for clear, modular monolithic codebases, all too often these arbitrary in-process boundaries break down
  - Code related to similar functions becomes spread all over, making fixing bugs or implementations more difficult
  - Cohesion – having related code grouped together – is an important concept associated with microservices → focus service boundaries on business boundaries, making it obvious where code lives for a given piece of functionality and avoiding the temptation for it to grow too large
  - The smaller the service, the more you maximize the benefits and downsides of microservice architecture

What are Microservices? *(2)*

- Autonomous
  - A microservice is a separate and isolated entity
  - Avoid packing multiple services onto the same machine
  - Although this isolation can add some overhead, the resulting simplicity makes it much easier to reason about
  - These services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change
  - A service exposes an application programming interface (API), and collaborating services communicate with it via those APIs; pick technology-agnostic APIs to ensure that the service does not constrain technology choices
  - To do decoupling well, you will need to model your services right and get the APIs right

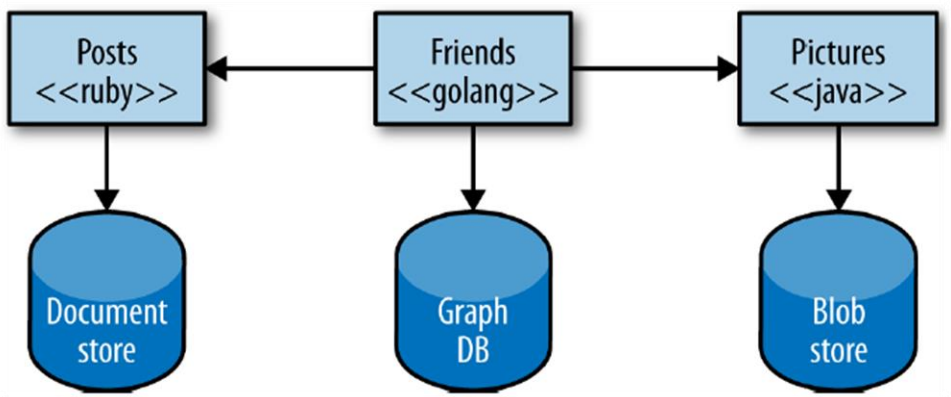
Key Benefits

Many of the microservices' benefits can be laid at the door of any distributed system; however, microservices tend to achieve these benefits to a greater degree

- Technology heterogeneity
- Resilience
- Scaling
- Ease of deployment
- Organizational alignment
- Composability
- Optimizing for replaceability

Technology Heterogeneity *(1)*

- This allows us to pick the right tool for each job



  - E.g.: for a social network, we might store our users' interactions in a graph-oriented database, but the posts the users make could be stored in a document-oriented data store

- We are also able to adopt technology more quickly and understand how new advancements may help us
  - Trying a new programming language, database, or framework on a monolithic system will impact a large amount of the system
  - With a system consisting of multiple services, we can pick a service that is lowest risk and use the technology there, limiting any potential negative impact

Technology Heterogeneity *(2)*

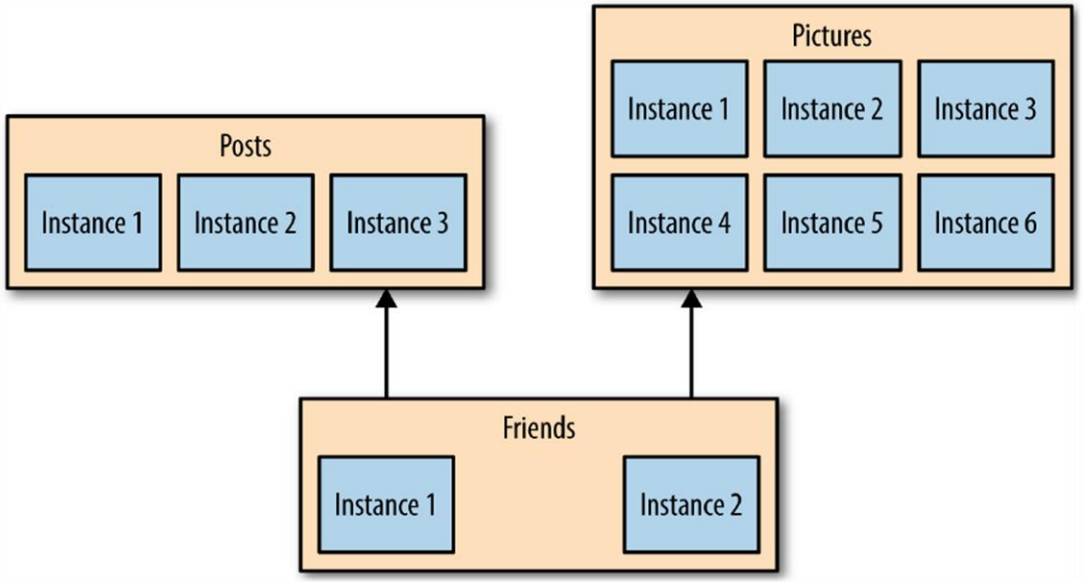
- Embracing multiple technologies does not come without an overhead, of course
  - Netflix and Twitter mostly use the Java Virtual Machine (JVM) as a platform, as they have a very good understanding of the reliability and performance of that system
  - They also develop libraries and tooling for the JVM that make operating at scale much easier, but make it more difficult for non-Java-based services or clients
  - But neither Twitter nor Netflix use only one technology stack for all jobs
- A counterpoint to concerns about mixing in different technologies is the size; if you really can rewrite a microservice in two weeks, you may well mitigate the risks of embracing new technology

Resilience

- A key concept in resilience engineering is the bulkhead; if one component of a system fails, but that failure does not cascade, you can isolate the problem and the rest of the system can carry on working
- Service boundaries become your obvious bulkheads
  - With a monolithic system, we can run on multiple machines to reduce our chance of failure
  - With microservices, we can build systems that handle the total failure of services and degrade functionality accordingly
- Networks can and will fail, as will machines; we need to know how to handle this and what impact (if any) it should have on the end user of our software

Scaling *(1)*

- With a large, monolithic service, we have to scale everything together; If one small part is constrained in performance and locked up in the giant monolithic app, we have to handle scaling everything as a piece



- With smaller services, we can <u>just scale those services that need scaling</u>, allowing us to run other parts of the system on smaller, less powerful hardware

Scaling *(2)*

- Examples:
  - Gilt, an online fashion retailer, started in 2007 with a monolithic Rails app, by 2009 Gilt's system was unable to cope with the load being placed on it. By splitting out core parts of its system, Gilt was better able to deal with its traffic spikes, and today has over 450 microservices, each one running on multiple separate machines.
  - When embracing on-demand provisioning systems like those provided by AWS, we can even apply this scaling on demand for those pieces that need it. This allows us to control our costs more effectively. It's not often that an architectural approach can be so closely correlated to an almost immediate cost savings.

Ease of Deployment

- A one-line change to a million-line-long monolithic app requires the whole app to be deployed in order to release the change → large-impact, high-risk deployment
- With microservices, we can make a change to a single service and deploy it independently of the rest of the system
  - Get our code deployed faster
  - If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve
  - E.g.: Amazon and Netflix use these architectures to ensure they remove as many impediments as possible to getting software out the door

Organizational Alignment

- Many problems are associated with large teams and large codebases; these problems can be exacerbated when the team is distributed
- Smaller teams working on smaller codebases tend to be more productive
- Microservices allow us to better align our architecture to our organization, helping us minimize the number of people working on any one codebase to hit the sweet spot of team size and productivity
- We can also shift ownership of services between teams to keep people working on one service colocated

Composability

- One of the key promises of distributed systems and SOA is that we open up opportunities for reuse of functionality
- With microservices, we allow for our functionality to be consumed in different ways for different purposes
  - We need to think of the myriad ways that we might want to weave together capabilities for the Web, native app, mobile web, tablet app, or wearable device
  - As organizations move away from thinking in terms of narrow channels to more holistic concepts of customer engagement, we need architectures that can keep up
  - We open up seams in our system that are addressable by outside parties

Optimizing for Replaceability

- Some big, nasty legacy system is difficult to replace; it's too big and risky a job
- With our individual services being small in size, the cost to replace them with a better implementation, or even delete them altogether, is much easier to manage
- Teams using microservice approaches are comfortable with completely rewriting services when required, and just killing a service when it is no longer needed
  - When a codebase is just a few hundred lines long, it is difficult for people to become emotionally attached to it, and the cost of replacing it is pretty small

What about SOA? *(1)*

- Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some end set of capabilities
  - A service here typically means a completely separate operating system process
  - Communication between these services occurs via calls across a network rather than method calls within a process
- SOA is …
  - An approach to combat the challenges of the large monolithic apps
  - An approach that aims to promote the reusability of software
  - An approach that aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service do not change too much

What about SOA? *(2)*

- Despite many efforts, there is a lack of good consensus on how to do SOA well
  - Much of the industry has failed to look holistically enough at the problem and present a compelling alternative to the narrative set out by various vendors in this space
  - Many of the problems are actually problems with things like communication protocols (e.g., SOAP), vendor middleware, a lack of guidance about service granularity, or the wrong guidance on picking places to split your system
  - SOA does not help you understand how to split something big into something small, how big is too big, and practical ways to ensure that services do not become overly coupled
- ☞ The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well; so, think of microservices as a specific approach for SOA

Decompositional Technique: Shared Libs *(1)*

- A very standard decompositional technique that is built into virtually any language is breaking down a codebase into multiple libraries; these libraries may be provided by third parties, or created in your own organization
- Teams can organize themselves around these libraries, and the libraries themselves can be reused
- There are some drawbacks:
  - You lose true technology heterogeneity; the library typically has to be in the same language, or at the very least run on the same platform
  - The ease with which you can scale parts of your system independently from each other is curtailed

Decompositional Technique: Shared Libs *(2)*

- There are some drawbacks: (cont'd)
  - Unless you are using dynamically linked libraries, you cannot deploy a new library without redeploying the entire process, so your ability to deploy changes in isolation is reduced
  - You lack the obvious seams around which to erect architectural safety measures to ensure system resiliency
- Creating code for common tasks that you want to reuse across the organization is an obvious candidate for becoming a reusable library; however, shared code used to communicate between services can become a point of coupling

Decompositional Technique: Modules *(1)*

- Some languages provide modular decomposition techniques that go beyond simple libraries
- They allow some lifecycle management of the modules, such that they can be deployed into a running process, allowing you to make changes without taking the whole process down
- Examples:
  - Open Source Gateway Initiative (OSGI), which emerged as a framework to allow plug-ins to be installed in the Eclipse Java IDE, is now used as a way to retrofit a module concept in Java via a library
  - In Erlang, modules are baked into the language runtime, thus they can be stopped, restarted, and upgraded without issue; Erlang even supports running more than one version of the module at a given time, allowing for more graceful module upgrading

Decompositional Technique: Modules *(2)*

- Problems with OSGI:
  - It is trying to enforce things like module lifecycle management without enough support in the language itself, resulting in more work having to be done by module authors to deliver on proper module isolation
  - Within a process boundary, it is also much easier to fall into the trap of making modules overly coupled to each other, causing all sorts of problems
- Problems with Erlang:
  - Limited in our ability to use new technologies
  - Limited in how we can scale independently
  - Can drift toward integration techniques that are overly coupling
  - Lack seams for architectural safety measures

No Silver Bullet

- Microservices are no free lunch or silver bullet, and make for a bad choice as a golden hammer
- If you are coming from a monolithic system point of view, you will have to get much better at handling deployment, testing, and monitoring to unlock the benefits; you will also need to think differently about how you scale your systems and ensure that they are resilient
- As every company, organization, and system is different, a number of factors will play into whether or not microservices are right for you, and how aggressive you can be in adopting them

Evolutionary Architect *(1)*

- Core responsibilities:
  - *Vision* – ensure there is a clearly communicated technical vision for the system that will help your system meet the requirements of your customers and organization
  - *Empathy* – understand the impact of your decisions on your customers and colleagues
  - *Collaboration* – engage with as many of your peers and colleagues as possible to help define, refine, and execute the vision
  - *Adaptability* – make sure that the technical vision changes as your customers or organization requires it
  - *Autonomy* – find the right balance between standardizing and enabling autonomy for your teams
  - *Governance* – ensure that the system being implemented fits the technical vision

Evolutionary Architect *(2)*

- The evolutionary architect is one who understands that pulling off this feat is a constant balancing act
- The worst reaction to all these forces that push us toward change is to become more rigid or fixed in our thinking
- Microservices give us many more decisions to make; therefore, being better able to balance all of these trade-offs is essential

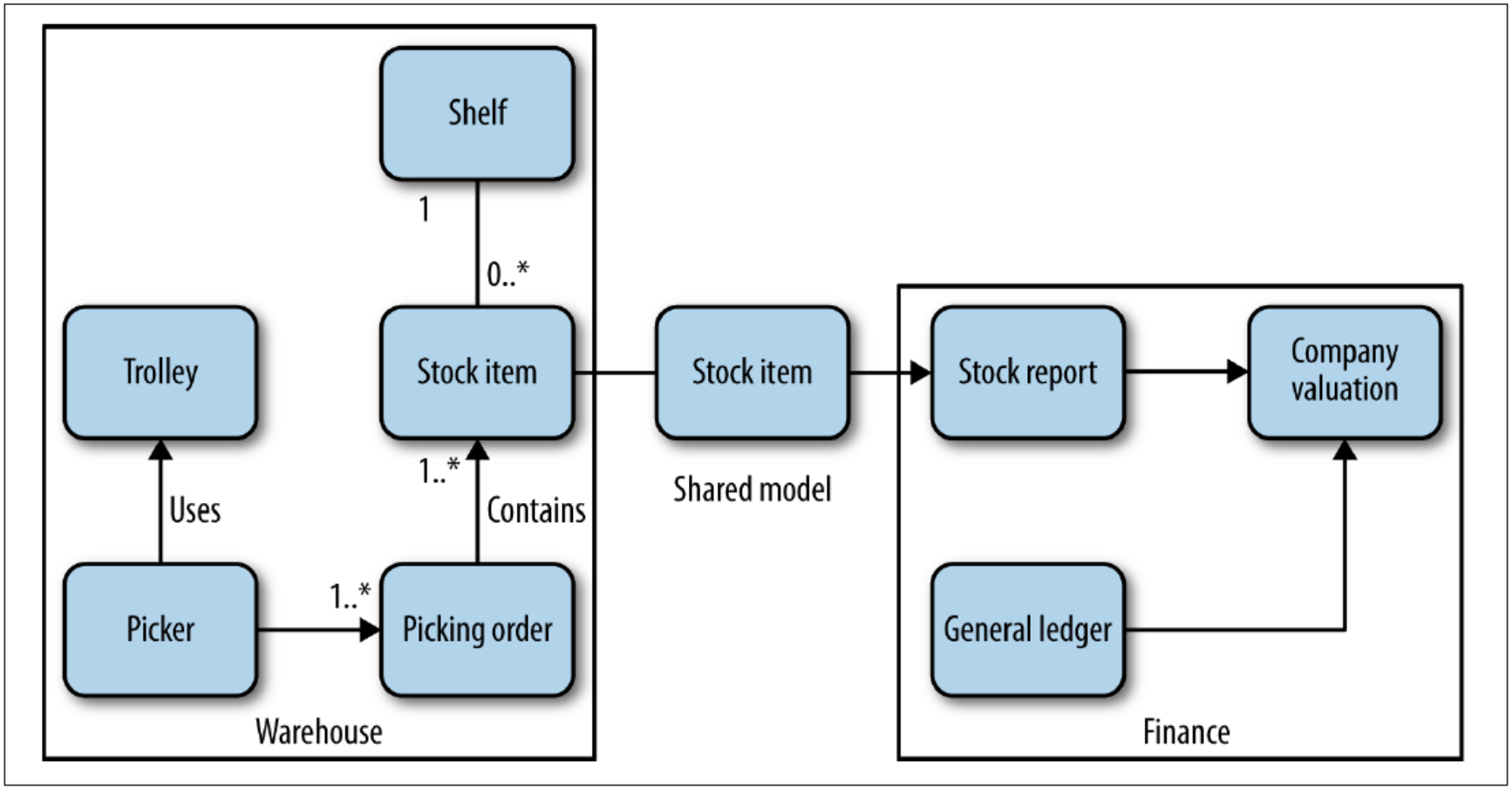
What Makes a Good Service?

- Loose coupling
  - When services are loosely coupled, a change to one service should not require a change to another
  - A loosely coupled service knows as little as it needs to about the services with which it collaborates
  - This also means to limit the number of different types of calls from one service to another, because beyond the potential performance problem, chatty communication can lead to tight coupling
- High cohesion
  - Place related behavior to sit together, and unrelated behavior to sit elsewhere
  - If we want to change behavior, we want to be able to change it in one place and release that change as soon as possible
  - Making changes in lots of different places is slower, and deploying lots of services at once is risky

Bounded Context

- Any given domain consists of multiple bounded contexts
- Residing within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts
- Each bounded context has an explicit interface, where it decides what models to share with other bounded contexts
- If you want information from a bounded context, or want to make requests of functionality within a bounded context, you communicate with its explicit boundary using models

Example of a Shared Model *(1)*



Example of a Shared Model *(2)*

- The finance department and the warehouse are two separate bounded contexts
- They both have an explicit interface to the outside world (in terms of inventory reports, pay slips, etc.), and they have details that only they need to know about (forklift trucks, calculators, etc.)
- To be able to work out the valuation of the company, the finance employees need info about the stock; the stock item then becomes a shared model between the two contexts

Example of a Shared Model *(3)*

- Note that we do not need to blindly expose everything about the stock item from the warehouse context
  - E.g.: a record on where a stock item should live within the warehouse needs not be exposed in the shared model
  - So, there is the internal-only representation and the external representation
- By thinking clearly about what models should be shared, and not sharing our internal representations, we avoid one of the potential pitfalls that can result in tight coupling

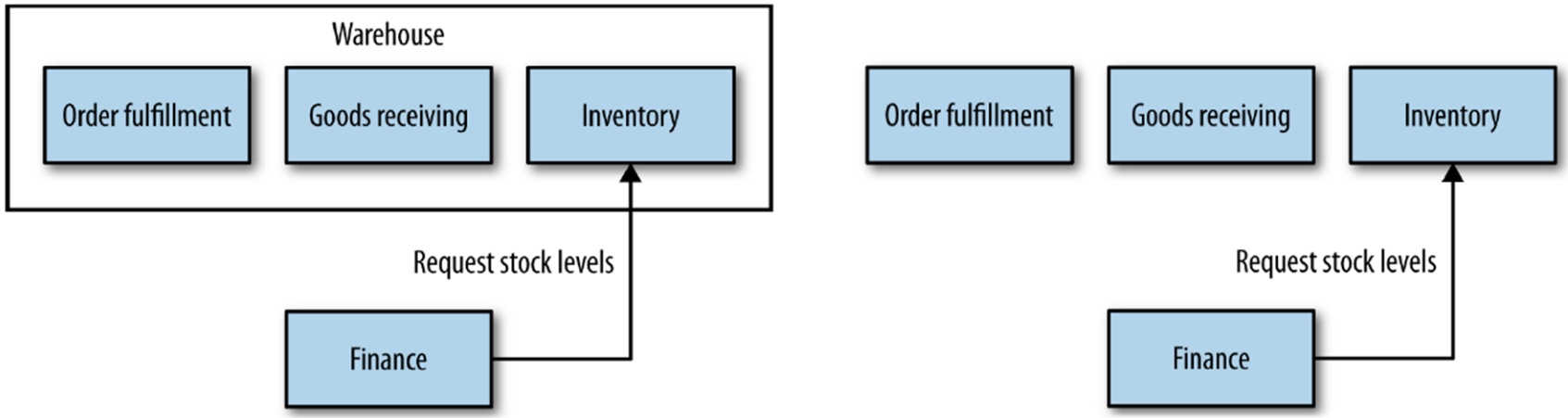
Business Capabilities

- When you start to think about the bounded contexts that exist in your organization, you should be thinking not in terms of data that is shared, but about the capabilities those contexts provide the rest of the domain
- Ask these two questions:
  - What does this context do?
  - What data does it need to do that?
- When modeled as services, these capabilities become the key operations that will be exposed over the wire to other collaborators

Decompose All the Way Down *(1)*

- At the start, you will probably identify a number of coarse-grained bounded contexts, but these bounded contexts can in turn contain further bounded contexts
  - E.g.: the warehouse could be decomposed into capabilities associated with order fulfillment, inventory management, and goods receiving
  - The outside parties are still making use of business capabilities in the warehouse and unaware that their requests are actually being mapped transparently to two or more separate services
- Sometimes, it makes more sense for the higher-level bounded context to not be explicitly modeled as a service boundary, so you might instead split out inventory, order fulfillment, and goods receiving

Decompose All the Way Down *(2)*



- There is not a hard-and-fast rule as to what approach makes the most sense; however, whether you choose the nested approach over the full separation approach should be based on your organizational structure

Comm. in Terms of Business Concepts

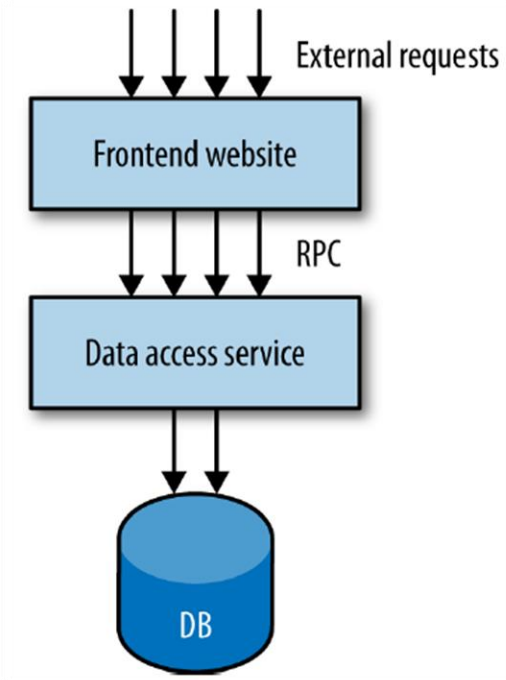
- If our systems are decomposed along the bounded contexts that represent our domain, the changes we want to make are more likely to be isolated to one, single microservice boundary
  - Reduce the number of places we need to make a change
  - Allow us to deploy that change quickly
- It is also important to think of the communication between these microservices in terms of the same business concepts
  - The same terms and ideas that are shared between parts of your organization should be reflected in your interfaces
  - It can be useful to think of forms being sent between these microservices, much as forms sent around an organization

Technical Boundary *(1)*

- In can be useful to look at <u>what can go wrong when services are modeled incorrectly</u>:
  - A company had a public-facing application with a large, global customer base
  - The system had taken on more features and more users
  - The company decided to have a new group of developers based in Brazil to take on some of the work
  - The system got split up (essentially, the repository layer in the codebase was taken and made into a separate service)
    - The front half of the app being essentially stateless, implementing the public-facing website
    - The back half of the system was simply a remote procedure call (RPC) interface over a data store

Technical Boundary *(2)*

- In can be useful to look at … : (cont'd)
  - Changes frequently had to be made to both services; they spoke in terms of low-level, RPC-style method calls, which were overly brittle
  - The service interface was also very chatty too, resulting in performance issues
  - It needed elaborate RPC-batching mechanisms; it had lots of layers (and was a painful exercise to cut through it)



- The team <u>picked what was previously an in-process API and made a horizontal slice</u>

Technical Boundary *(3)*

- On the face of it, the idea of splitting the previously monolithic system along geographical/organizational lines (e.g., a vertical, business-focused slice) makes perfect sense
- Making decisions to model service boundaries along technical seams is not always wrong; however, it should be your secondary driver for finding these seams, not your primary one