Designing Data Intensive Applica

## Building microservices Ch6

Workflow

What happens when we want multiple μservices to collaborate?

Database Transactions

≥1 operation to be done as a single unit. want to confirm changes have been made + a way to clean up in case of an error

ACID Transactions

Atomicity, Consistency, Isolation, Durability

Atomicity	Consistency	Isolation	Durability
all actions complete or fail	db is in a valid state after changes	∞ txns operate w/o interfering w/ each other.	once txn has completed, data won't get lost in the event of a system failure

* transactions within a single μservice can be ACID transactions, but it gets more complex across μservices

"we have to accept that by decomposing this operation into two separate database transactions, we've lost guaranteed atomicity of the operation as a whole"

Distributed Transaction - 2-phase commits

Two-phase commit algorithm (aka 2PC) (↓)

① voting phase	② commit phase	
a central coordinator asks all workers in the txn if a state change can be made. All agree → proceed. ≥1 disagree → abort & send rollback message	send commit message to everyone, may be handled at different times so there may be a window of inconsistency · changes are made + locks released	* the worker is guaranteeing it can make the change at some point in the future & will likely lock the record

* you'll be coordinating locks among many participants & trying to avoid deadlocks

* handling failure states can be tricky & may require human intervention

* 2PC can ↑ latency in a system & is typically only used for very short lived operations, the longer ~~that~~ the operation, the longer your resources are locked.

* you can choose to leave a state change in your DB or monolith vs splitting it due to these distributed concerns

The limits of the saga pattern - ~~[illegible]~~ Uwe Friedrichsen

Sagas

- algo that can coordinate ∞ changes in state w/o locking resources for a long time by modeling the steps involved as discrete activities to be executed independently. Forcing us to define business processes explicitly.
- Built to address *long lived transactions* (LLTs) (minutes, hours or days) & also works for coordinating change across ∞ services

"We can break a single business process into a set of calls that will be made to collaborating services — this is what constitutes a saga"

- each section of a saga is an ACID txn but the overall process is not. It's up to the developers to handle the implications of this

Saga Failure Modes

Backward recovery: reverting a failure & cleaning up (rollback)
Forward recovery: pick up from where a failure occurred & continue
Both.

* allows a recovery from a business failure, not technical failure
(ex. insufficient funds) (ex 500 error)

* Sagas assume that the underlying system & components are reliable

* how to rollback an already committed transaction?
initiate a compensating transaction for each step that has already been committed. Note that we can't rollback time & pretend like the original commit never happened, we're only offsetting what was committed to "undo" it.

* These compensating transactions are semantic rollbacks because we do what we need to to clean up the saga. Ex. Confirmation email already sent but order was cancelled? Send cancellation email.

* need to persist rollback information

* The rollback process can be simplified if the steps are reordered, process actions that are more likely to fail earlier

Orchestrated Sagas, an orchestrator defines order of execution & trigger any required compensating actions. "Command & control" approach typically uses request/response. ↑ coupling & logic leak into orchestrators. The services are still entities w/ own local state & behavior

Practical Process Automation
- Bernd Ruecker

Enterprise Integration Patterns by Gregor Hohpe & Bobby Woolf

WARNING
" If logic has a place where it can be centralized, it will become centralized! "
- You can have different services orchestrating for different flows to avoid this

Business Process Modeling (BPM)(↓) are tools for non-devs to define business process flows. Although typically devs end up using it & the result is uncomfortable, so they don't recommend it.
Camunda & Zeebe

Choreographed Sagas
"trust but verify architecture" where operation of the saga is entrusted to multiple collaborating services & often use events heavily through a message broker
- ↓ domain coupling ↓ centralizing logic, but ↑ complexity, harder to work out what's happening.
- Difficult to know what state a saga is in as well
  ↳ can be offset by using a unique ~~[illegible]~~ correlation ID for the saga & have a service to collect the events for that correlation ID & show the state of the saga

Mixing Styles
- some business cases naturally fit into one style of saga or another, in other cases you may need to mix & match approaches
"if you do decide to mix styles, it's important that you still have a clear way to understand what state a saga is in and what activities have already happend..."

* if one team owns implementation of the entire saga it's easier to use orchestrated sagas, if not, lean towards choreographed sagas.