Building Microservices Ch5

Implementing μservice comms

Decide b/w

- blocking sync
- nonblocking async
- request-response
- event-driven collab

to help inform the technology to choose

The tools should:

- make backward compatibility easy
  - don't break ↓stream μservices
  - we can validate this before deploying μservice to prod
- make your interface explicit
  - it's clear to consumers what the μservice can do
  - " " " does " functionality needs to remain intact
  - the suggest an explicit schema + supporting docs
- Keep your APIs technology Agnostic
  - the way you comm b/w μservice technology agnostic
- Make Your Service Simple for Consumers
  - in a way that makes sense for your use case, ex giving consumers full tech choices or implementing a library for them to use
- Hide internal implementation detail
  - avoid tech that pushes us to expose internal representation

* Here highlighting some of the most popular & interesting. In particular:
  * Remote Procedure Calls (ex. SOAP / gRPC)
  * REST
  * GraphQL
  * Message brokers

Remote Procedure Calls (RPC)

* making a local call + having it execute on a remote service
- most tech requires an explicit schema (ex. SOAP | gRPC)
  ↳ aka Interface Def Lang (IDL)
  or for SOAP, Web Service Def Lang (WSDL)
- all make a remote call look local
- You're typically buying into a serialization protocol w/these tools
  (ex. gRPC uses protocol buffer serialization format
  ↳ or tied to a specific networking protocol (ex. SOAP w/HTTP)
- the client can typically generate code using the given schema
  so long as it has access to the schema before it needs to make a call
  ex. Avro RPC can send full schema + payload so that clients can dynamically interpret the schema

Challenges

- tech coupling (ex. Java RMI is tied to a platform)
  (Thrift + gRPC has wide lang support)
- can come w/restrictions on interoperability
  ↳ except for gRPC, SOAP + Thrift for example
- local calls aren't like remote calls
  * cost can be ↑ + time can be ↑
  * if too ↑ abstraction, devs may not even know they're making remote calls

"You should assume that your networks are plagued w/malevolent entities ready to unleash their ire on a whim"
↳ can you tell the diff. btw diff. failures?

- Brittleness
  - changes to your schema would require client updates for even small changes + they're common
  * you can't separate client + server deployments *
  - what if you remove a field (ex. age from Consumer obj.)?
    All client objs. would need to be updated, or they'd break
  - think of these objs. as "expand-only"

$ REST in Practice Hypermedia & Systems Architecture (O'Reilly) by Jim Webber, Savas Parastatidis & Ian Robinson

Richardson maturity model

Where To Use RPC?

*gRPC recommended

REST (Representational State Transfer)

- Resources: a thing that the service itself knows about
- REST is commonly used over HTTP because HTTP verbs make implementing REST over HTTP easier
  - ↳ verbs (ex GET, POST, etc) should act the same on all resources
- conceptually there is one endpoint & the verbs act over that endpoint
- You can do gRPC over HTTP too!
- But you have to use HTTP well, otherwise it won't scale
- ★ Hypermedia as the engine of application state (HATEOAS)
  - ↳ hypermedia: concept where a piece of content contains links to various other pieces of content in a variety of formats (ex. text, images, sounds)

I don't totally understand this. Need more research

HATEOAS: clients should perform interactions with the server via these links to other resources.

"As long as these implicit contracts b/w the customer & the website are still met, changes don't need to be breaking changes"

"I need to access the resource, find the buy control, and navigate to that" to decouple client & server

"Fundamentally, many of the ideas in REST are predicated on creating distributed hypermedia systems & this is not what most people end up building"

Challenges

- can't typically generate client libraries, so you end up writing some
- OAS can help, but hasn't really been adopted & there is a big difference b/w documenting an API & creating an explicit contract with it
- ↓ efficiency over things like SOAP/HTTP or Thrift + REST is locked in w/ TCP? But HTTP/3 is looking to address these limitations w/ QUIC

Rest cont'd

Where To Use It

- if you're looking to allow access from a wide variety of clients
- it's widely understood & has ↑ interoperability
- good for interactions outside of the μservices
  - for sync. request-response based comms b/w μservices

GraphQL

- makes it possible for a client-side device to define queries that avoid the need to make multiple requests to receive the same info.
- ↓ # calls & info in those calls to help w/ use cases like mobile
→ a single query can pull all req'd info
→ need a μservice to expose a GraphQL endpoint
→ the endpoint exposes a schema
→ the schema exposes the types available
  + can use a graphical query builder to build these queries

Challenges

① client can do dynamically changing queries which can ↑ load on server
  + can be hard to pin down

② Caching is more complex, you can associate an ID w/ every returned resource + the client can cache the request against the ID
  ↳ so can't use CDNs or caching reverse proxies w/o addtl work
  ↳ may end up w/ REST API for generic requests & GraphQL for other requests

③ Not well suited for writes
  can make it feel like μservices are just wrappers for a DB when they actually expose functionality over a network
  ↳ need to make sure your GraphQL API isn't coupled w/ your datastore

Where To Use It

- perimeter, exposing functionality to external clients
  ↳ typically GUIs or mobile devices
- or external API if clients have to make many calls
- it's a call aggregation & filtering mechanism so it wouldn't replace general μservice-μservice comms
  ★ Can alternatively look into Backend For Frontend (BFF) pattern

Designing Event-Driven Systems by Ben Stopford

Kafka: The Definitive Guide by Neha Narkhede, Gwen Shapira + Todd Palino

Message Brokers

- are intermediaries aka middle ware
- manage comms b/w μservices for async comms
- message: generic concept that defines the thing a message broker sends
  - ↳ could contain a request, response, or event
- μservice gives mesage to message broker w/ info abt how to send the msg.
- queue: typically point-point, publish + consume
- topic ∞:1, many subscribers · 1 topic, all subs get msg
- Consumer could be ≥1 μservice, typically modeled as a consumer group
  for ex. if you have many instances of a μservice + any should receive msg so the queue can work as a load distribution mechanism + is an example of the competing consumer pattern
- a queue knows where a message is heading, a topic doesn't

Guidance

Topic: event-based collab

Queue: request/response comms

* ~~Guarantees~~
* Guaranteed Delivery
  - but read the docs, every msg broker has its limitations
  - also need to trust your tool + the people who created it
* can (maybe) guarantee order of delivery
* " provide transactions on writes (ex multiple writes at once)
* " " " " read, good to make sure a message is read by the consumer before removing it from the queue
* exactly once delivery, but double check how your broker guarantees this + write as if you'll receive multiple msgs anyway

Choices

- RabbitMQ, ActiveMQ, Kafka, Simple Queue Service (SQS) (AWS), Simple Notification Service (SNS) (AWS), kinesis

Highlight: Kafka

→ can help w/ more real-time processing bc of its stream process. pipeline

① ↑ scale ② ∞ consumers/producers in a cluster ③ msg permanence ④ built-in support for stream processing

Serialization Formats

- when given a choice on how we convert data for network calls, what to choose?

Text Formats

↑ JSON > ↓ XML (popularity), but JSON threw out schemas?

Avro, can send schema w/ payload

XML has XPATH standard or CSS selectors + lots of tooling support

Binary Formats

- if worried about ↑ payload or efficient reads/writes
- Protocol Buffers
- Simple Binary Encoding, Cap'n Proto, Flat Buffers

* for very ↓ latency needs

Schemas

- picking a serialization format will likely inform your schema tech

ex XML → XML Schema Def (XSD)

JSON → JSON Schema | SOAP → WSDL

gRPC → protocol buffer specification

* ↑ explicit schema

① goes a long way to being an explicit representation of what a μservice endpoint exposes + what it can accept

② help to catch accidental breakages

Breakages

2 categories: Structural + Semantic

Structural: endpoint structure changes in a way that the consumer is no longer compatible

Semantic: the behavior changes + breaks consumers' expectations

- Compare schema versions to catch structural breaks
- Test to catch semantic breaks

Change

- how to handle changes in μservices?

go to next page →

How to handle changes?

Avoid breaking changes:

Expansion changes: add new things, don't remove old things

Tolerant reader: Be flexible in what you expect from μservice

Right Technology: pick tech that makes being backwards compat. easier

Explicit Interface: makes it easier to tell what can be changed

Catch breaking changes early: have mechanisms in place to catch breaks as soon as possible

"This pattern — of implementing a reader able to ignore changes we don't care about — is what Martin Fowler calls a tolerant reader"

Postel's Law: (aka robustness principle)

"Be conservative in what you do,
be liberal in what you accept from others"

AsyncAPI & CloudEvents tried to help be explicit about which events are exposed by a μservice

Semantic Versioning:

→ MAJOR . MINOR . PATCH

++: breaking change

++: new func. still backwards compatible

bug fixes for existing functionality has been made

Catch breaks early w/ pass/fail tools that can find incompatible schemas in your CI build

→ Protolock, json-schema-diff-validator, openapi-diff

Confluent Schema Registry → catch structural breaks

→ Pact (an ex.) → to catch semantic breaks

Managing Breaking Changes:

Lockstep Deployment: require everything to be changed & deployed together

Coexist incompatible μservice versions: can coexist endpoints in same μservice instead of separate versioned instances

* Emulate old interface (author preferred)

Expand & Contract Strategy

Owner + consumer of μservice need to agree on:
① How will you raise that an interface needs to change?
② How will we collab. to agree on the change?
③ Who is expected to do the work to update the consumers?
④ when the change is agreed on, how long will consumers have to shift to the new interface before it is removed?

* try to embrace a consumer-first approach
* need to track usage of your endpoints

DRY: (Don't Repeat Yourself) more accurately means we want to avoid duplicating our system behavior + knowledge
↳ however, sharing code in μservices is a bit more complicated

Libraries
- can cause ~~co~~coupling b/w μservices that use the library
- if your shared code ever leaks outside your service boundary you've introduced a potential form of coupling
- libraries are typically packaged and deployed with each μservice so you'll need to redeploy them to use the new library version
- is it O.K. to have multiple versions of a library in the world?

Client Library
- if the server + client API are written by the same team there is a chance that server logic will leak into the client
  → you ↓ cohesion + ↑ tech choices if client library must be used
- can have SDKs written by anyone other than those that ~~use~~ write the server API
  ↳ need to ensure client is in charge of upgrades
- client libs also help w/ scalability + reliability. They handle service discovery, failure modes, logging + other non-service specific aspects

* be sure to separate handling transport protocol (help w/ service discovery + failure) from things related to the destination service itself
* will you force consumers to use the client library?
* maintain independent deployability, should be able to release independently

Service Discovery
① provide a mechanism for an instance to register itself
② provide a way to find the service once it's registered
→ should be able to handle instances being destroyed & created
Domain Name System (DNS)
Route53, Consul
- They have a TTL (time to live) & will be cached for at least that long
- Can use a load balancer to avoid stale instances if you have multiple instances of a host
Dynamic Service Registries
- μservices register & others can look in the registry to find what they need for more dynamic environments
Zookeeper (↓), Consul & Consul-template & Vault, etcd & Kubernetes, creating your own (↓)
- humans will also likely want to look at this information
Service Meshes & API Gateways
- we have the concept of a network perimeter
  - API Gateway: handle traffic inside & outside network perimeter
  - Service Mesh: handle traffic within network perimeter & blw μservices
- Service meshes & API Gateways can work as proxies blw μservices and handle μservice-agnostic behavior like service discovery or logging
- any shared behavior should be 100% generic
- API Gateway: mapping from ext. parties to internal μservices & are typically built on top of a simple HTTP Proxy & largely function as reverse proxies. Can also store API keys, logging, rate limiting, etc. Need one for Kubernetes but if you just need external third-party access it's overkill
Ambassador, but there's a lot of overselling so avoid tools that push for aggregation tools, protocol rewriting but we need to keep the pipes dumb & the endpoints smart
"The more behavior you leak into API gateways ... the more you run the risk of handoffs, increased coordination, and slowed delivery"

Service Meshes

- Common functionality w/ inter-service comms is pushed into the mesh
  ↳ mutual TLS, correlation IDs, service discovery, load balancing, etc
- its becoming an assumed part of any given platform created for self-service deployment & management of services
- (+) making it easy to implement common behavior & improve them w/o needing a redeploy
- should be running on the same system as the services so that what looks like a remote call is run locally & is faster
- A control plane sits on top of local mesh proxies where behavior can be changed & you can collect information
- Envoy proxy is commonly used & is a precursor for Istio + Ambassador. The proxies are controlled by a control plane to help see & control what's being done.
  Ex. Service Mesh implements mTLS, control plane distributes client & server certificates

* No business functionality has leaked into Service mesh, so it's still a dumb pipe

Linkerd, Istio

- options are limited on kubernetes & it adds complexity, if you want services to be written in different languages.

* switching b/w service meshes is painful!

Documenting Services

- Service discovery lets you know where things are, but how do you know what they are or how to use them? what they do?
- An explicit schema helps to document the structure & detail of the interface. But you still need to document why & how the endpoint is used
- OpenAPI, Ambassador's developer portal, Async API, Cloud Events, Backstage (Spotify)
- Can pull in health check & monitoring information as real-time documentation along w/ other metadata
- "Humane registries", human curated & readable?