Design patterns for microservice architecture

x

whoami

- Lead Software Developer
  @ The Software House
- adiqpl @ twitter
- adrian@zmenda.com
- linkedin.com/in/adrianzmenda

Adrian Zmenda

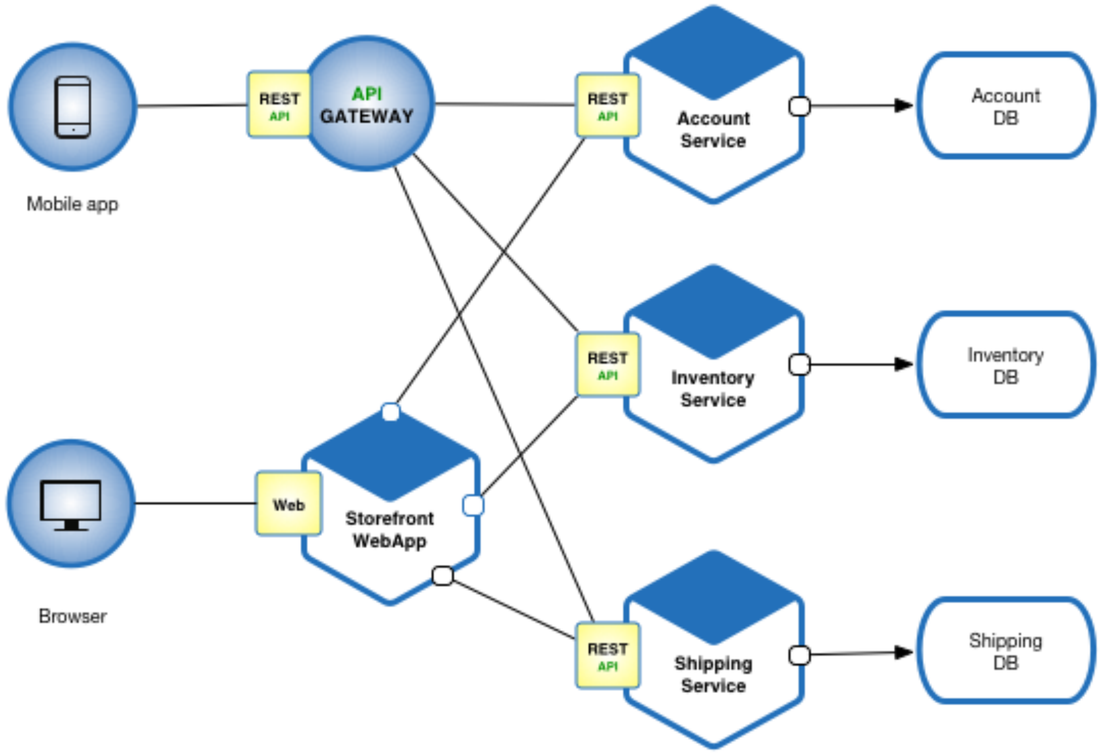
What we’ll cover today

- Basics
- Communication
- Internal Communication
- Security
- Availability
- Configuration
- Logging

Should I use microservices?

What is a microservice?

- Loosely coupled
- Independently deployable
- Highly maintainable and testable
- Organized around business capabilities
- Owned by a small team



When microservices are good idea

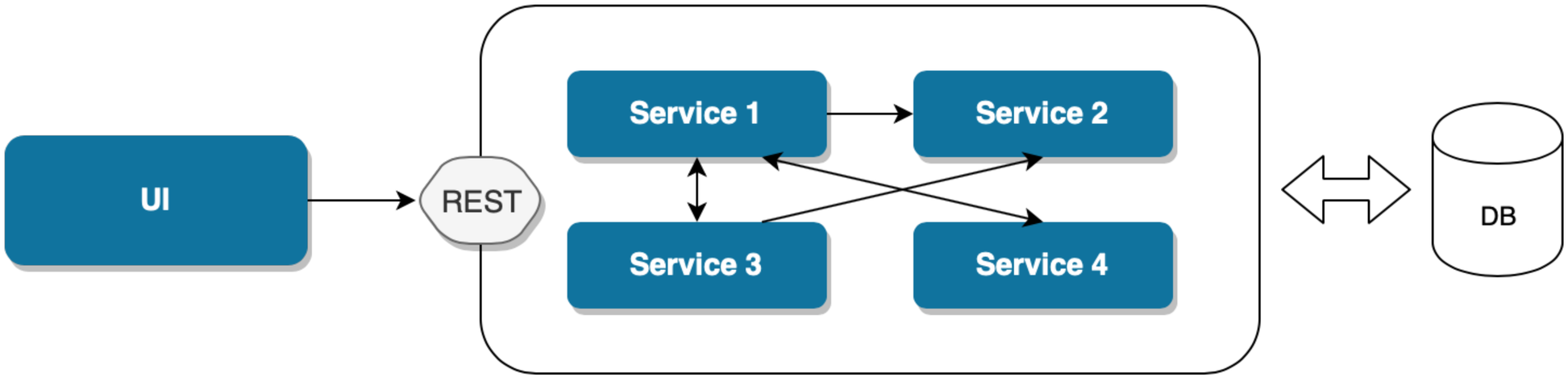
- Complex application
- High scalability
- Rapid and frequent delivery

Drawbacks

- running distributed system is **hard**
- maintaining consistency is **challenging**
- ops **complex**ity is high

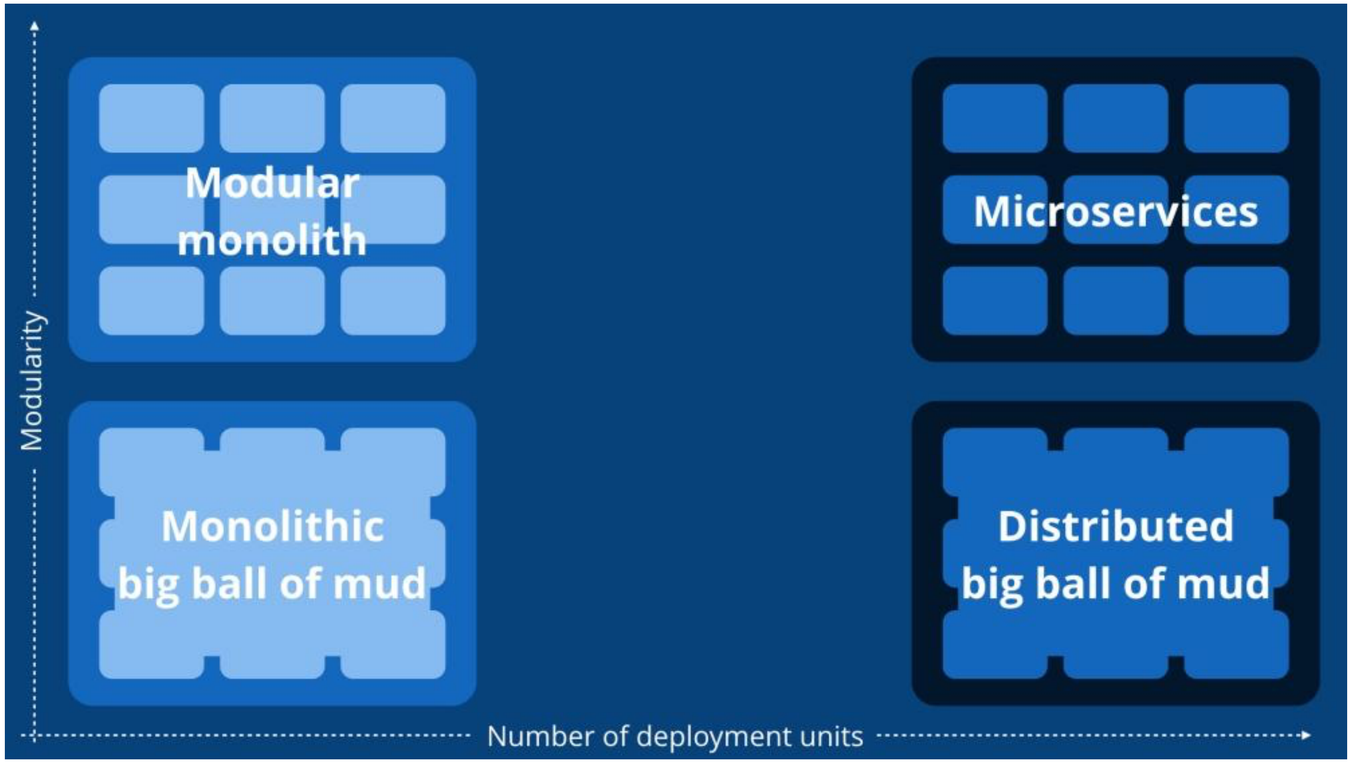
Should I start with microservices?

Monolith



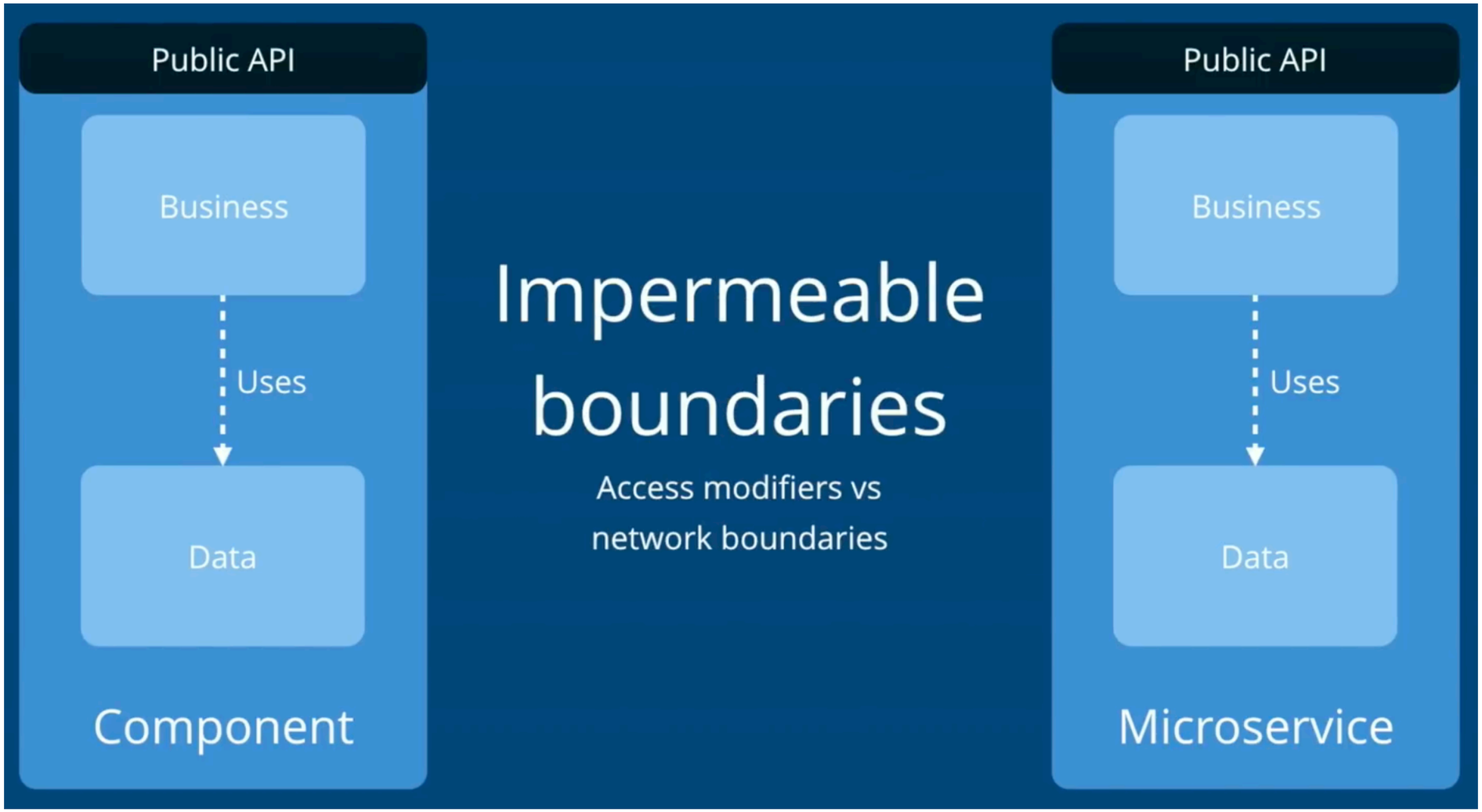
Monolith

source: CBS



source: Simon Brown

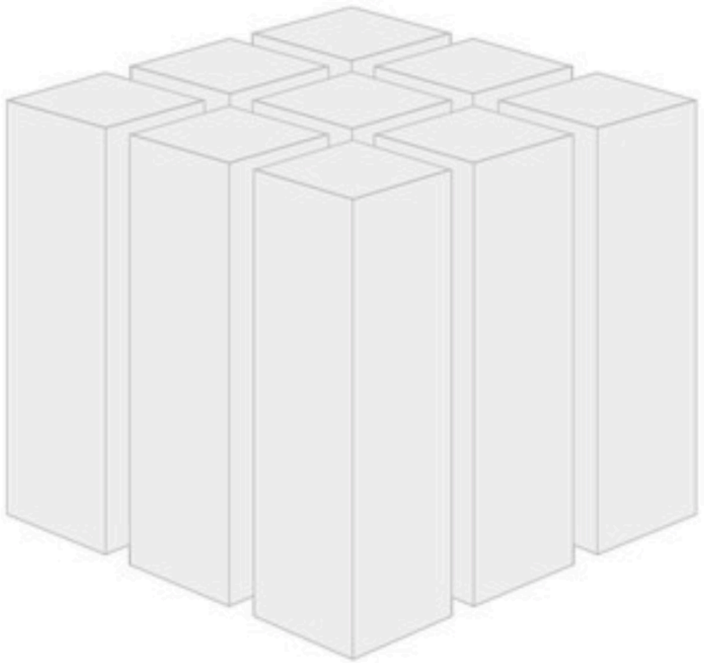
Modular Monolith

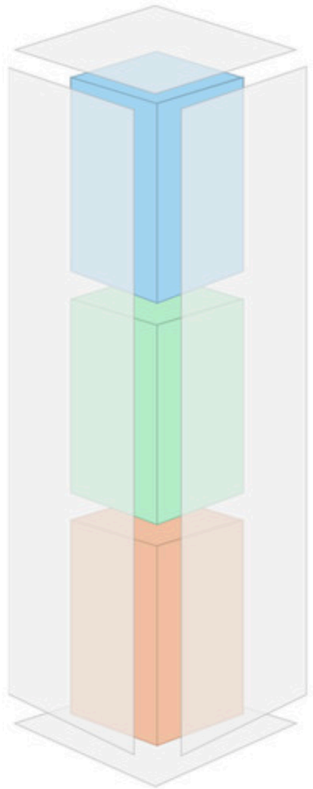


source: Simon Brown

Can we go even further?

Self-Contained System





An SCS contains its own user interface, specific business logic and separate data storage

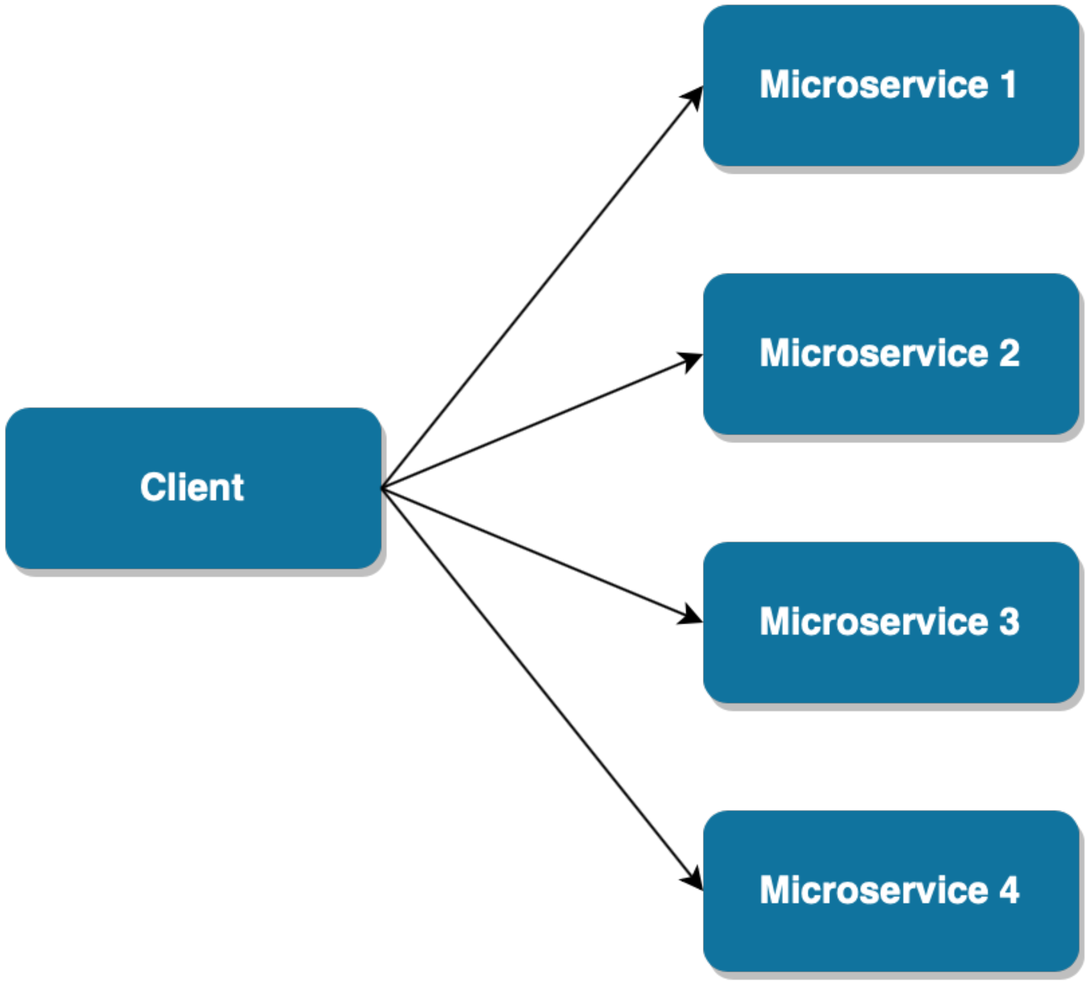
source: Roman Stranghöner

Self-Contained System

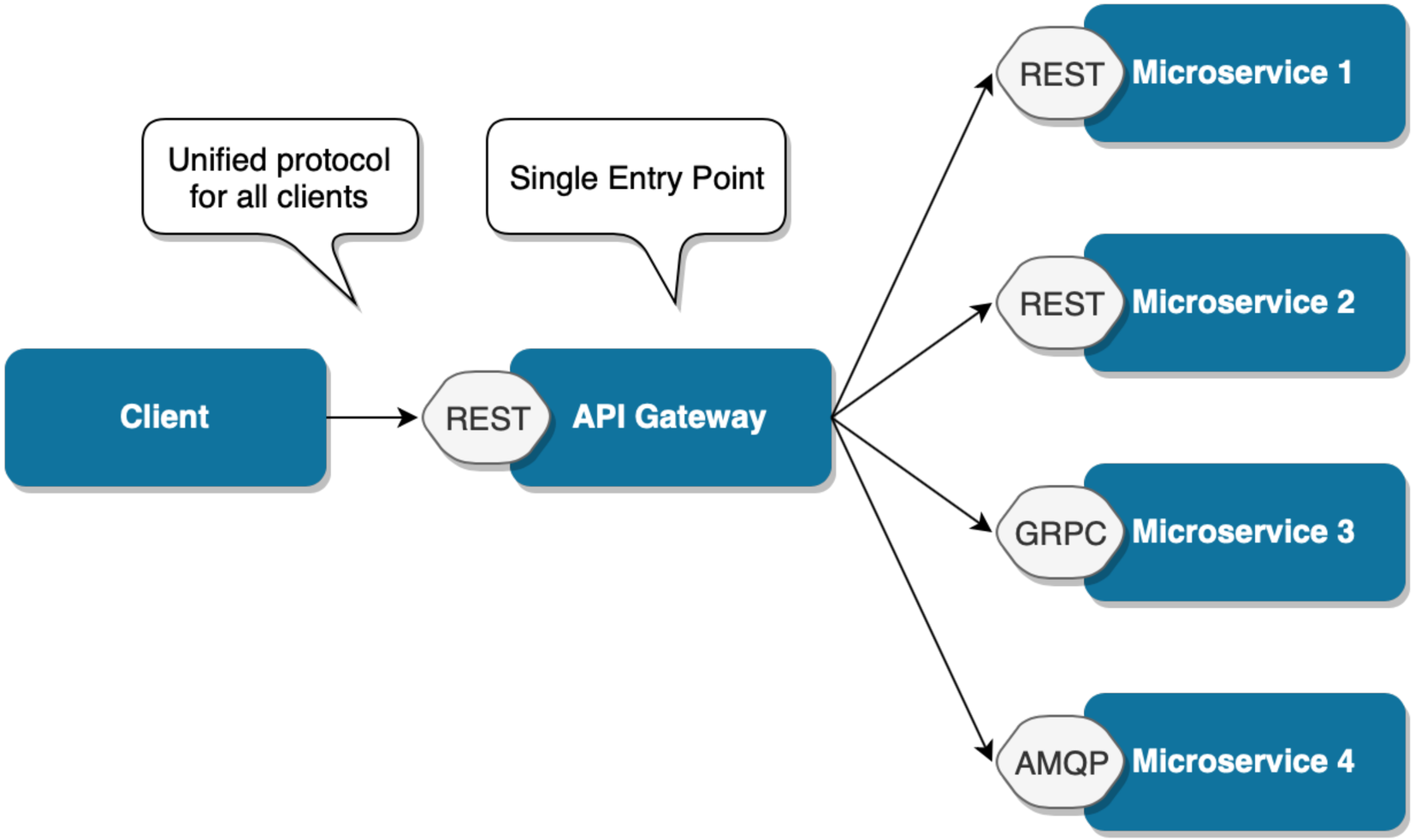
sooo, it’s like microservices. right?

Communication

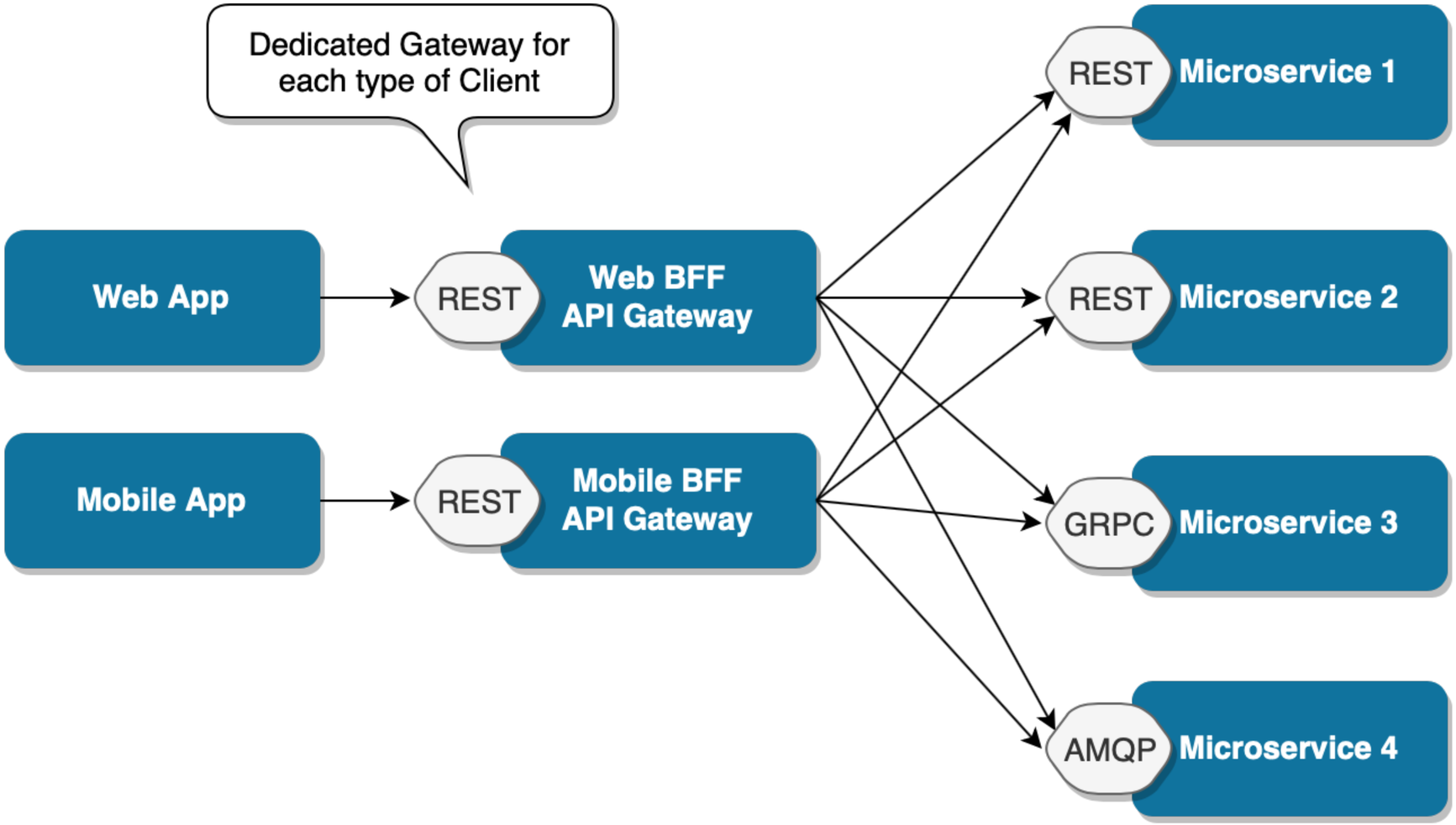
Direct


Client
Microservice 1
Microservice 2
Microservice 3
Microservice 4

Gateway


Unified protocol for all clients
Single Entry Point
Client
REST
API Gateway
REST
Microservice 1
REST
Microservice 2
GRPC
Microservice 3
AMQP
Microservice 4

Backend For Frontend



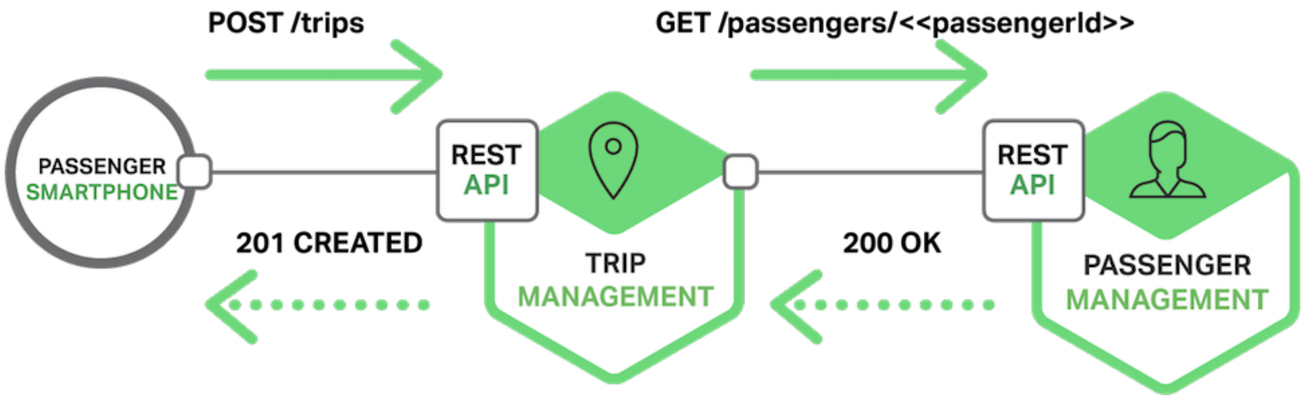
Internal Communication

Remote Procedure Invocation

- Synchronous
- Simple
- Easy to understand
- Follows Request / (Response)

REST

- Familiar
- Based on HTTP
- No intermediate broker



source: nginx

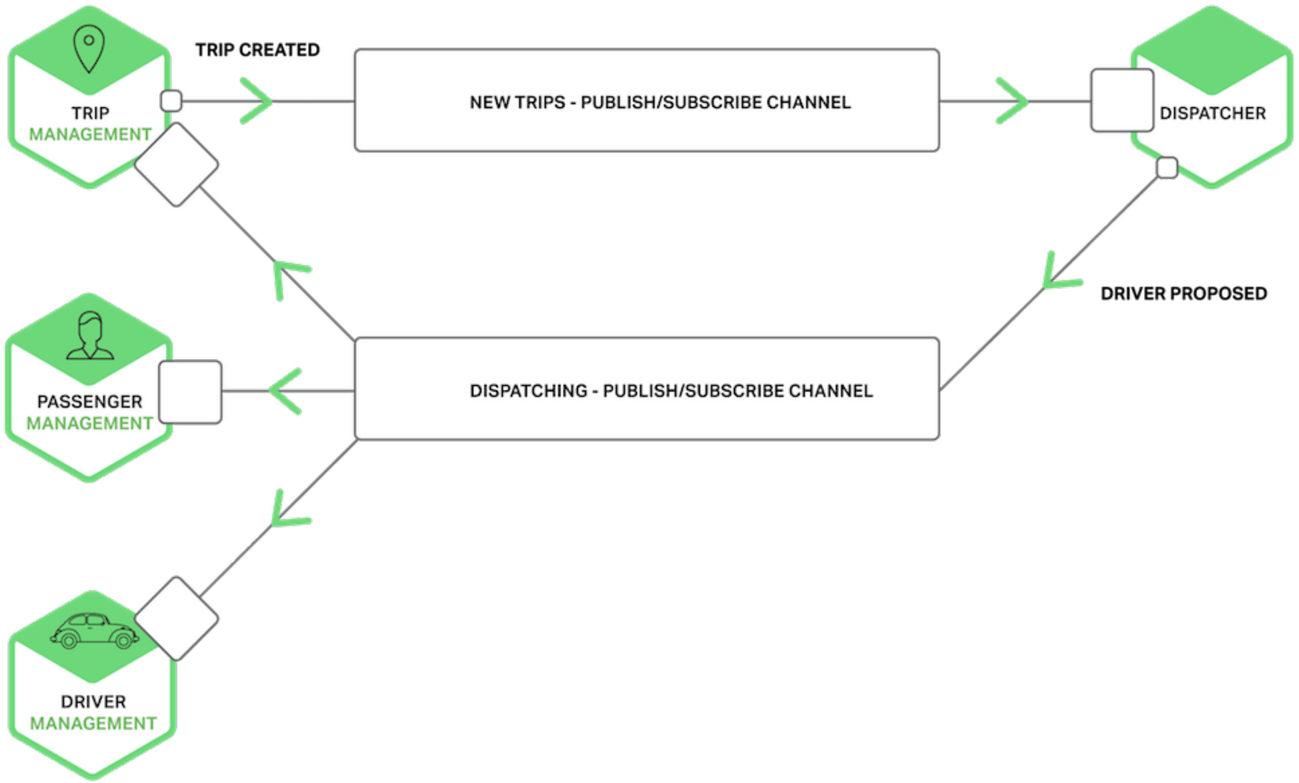
Apache Thrift

- Backed by Facebook
- Language neutral contract-file
- Code generation
- Support for notifications

- Backed by Google
- Calls visible as internal
- Code generation
- Simple contracts (Protocol Buffers)

Message Broker

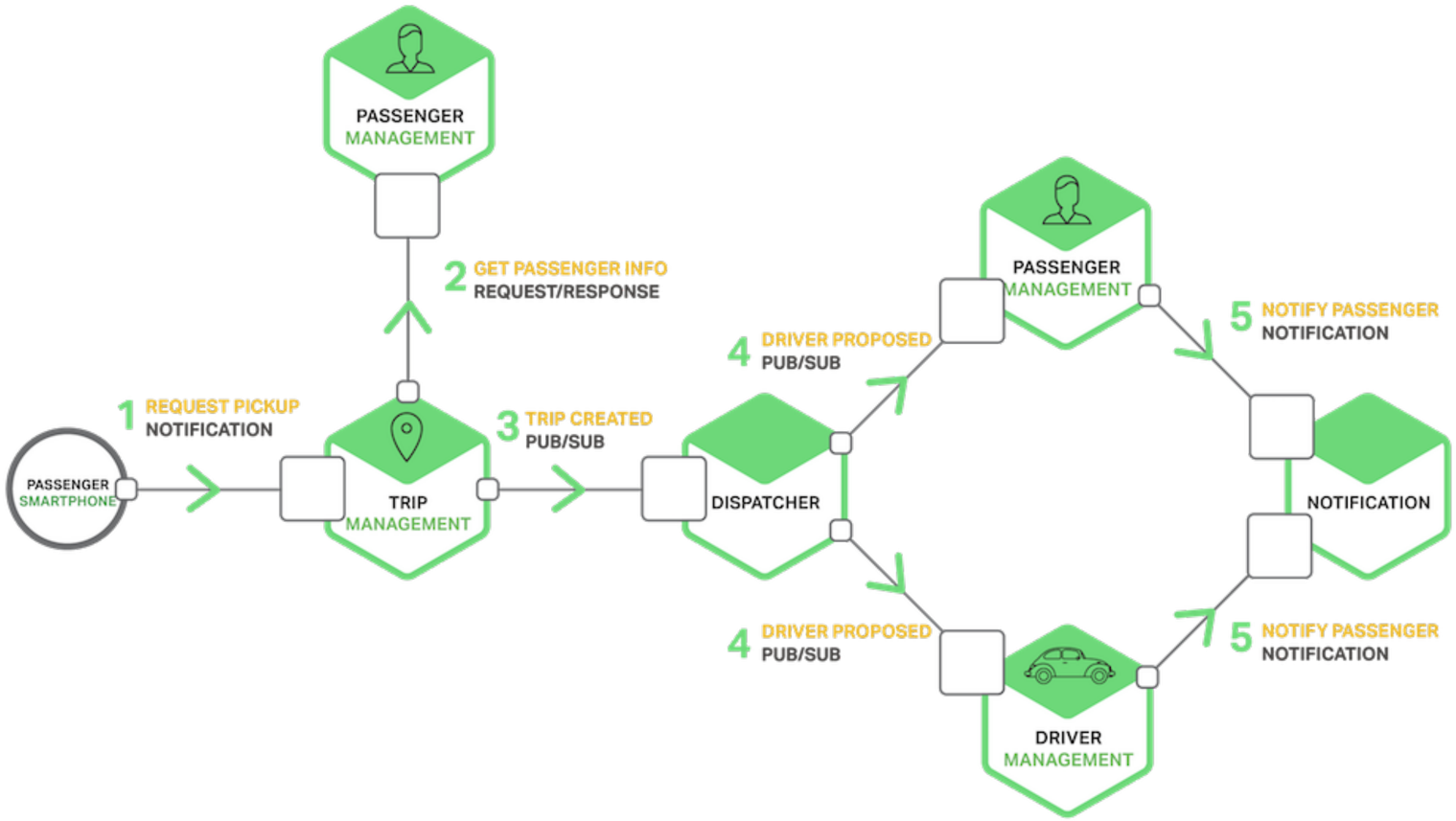
- Asynchronous
- Follows Publish / Subscribe
- Increases complexity



source: nginx

Message Broker

Combined Messaging



source: nginx

Security

No ACL

- Simple
- Rarely possible
- No additional over-head

ACL in every microservice

- Duplication of ACL logic
- Redistributes ACL across all microservices
- Contexts mixing

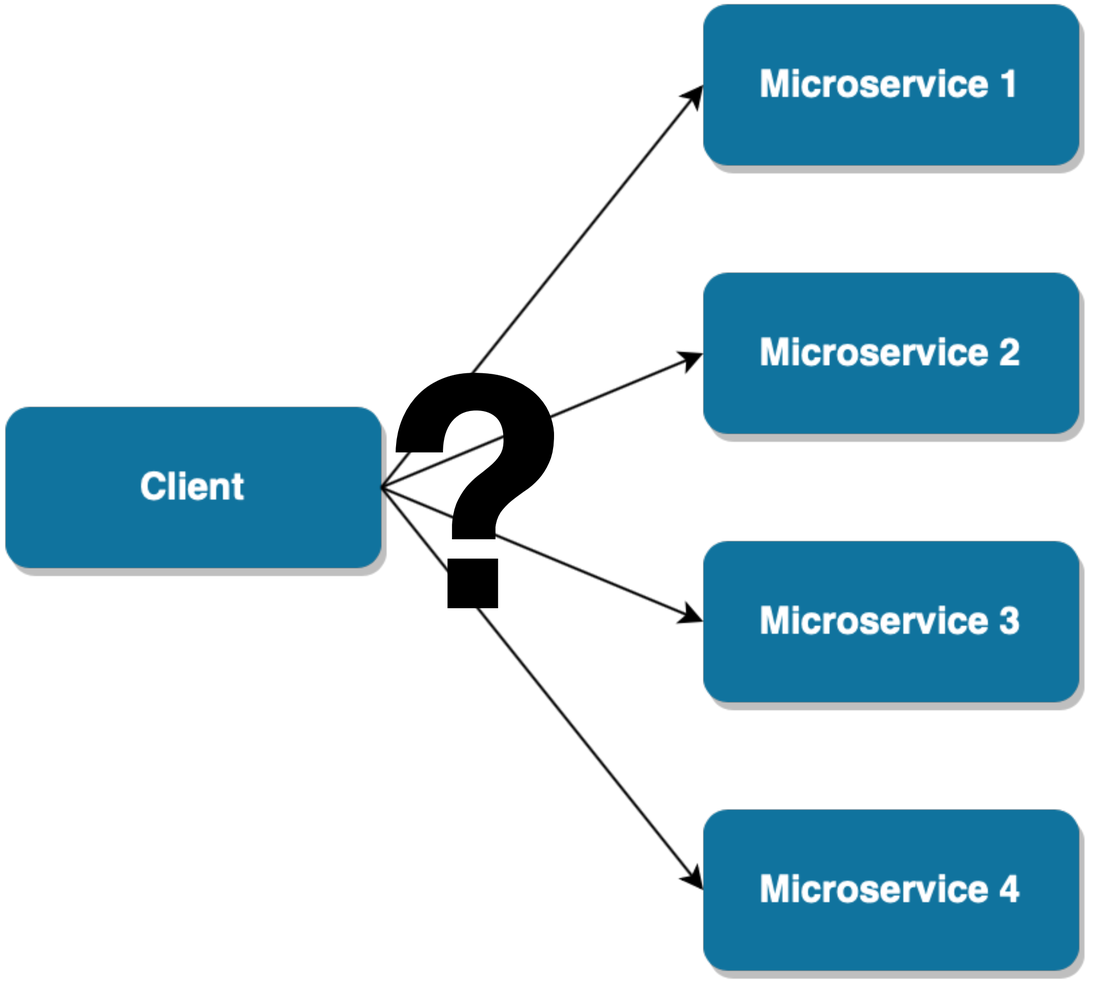
ACL as dependent microservice

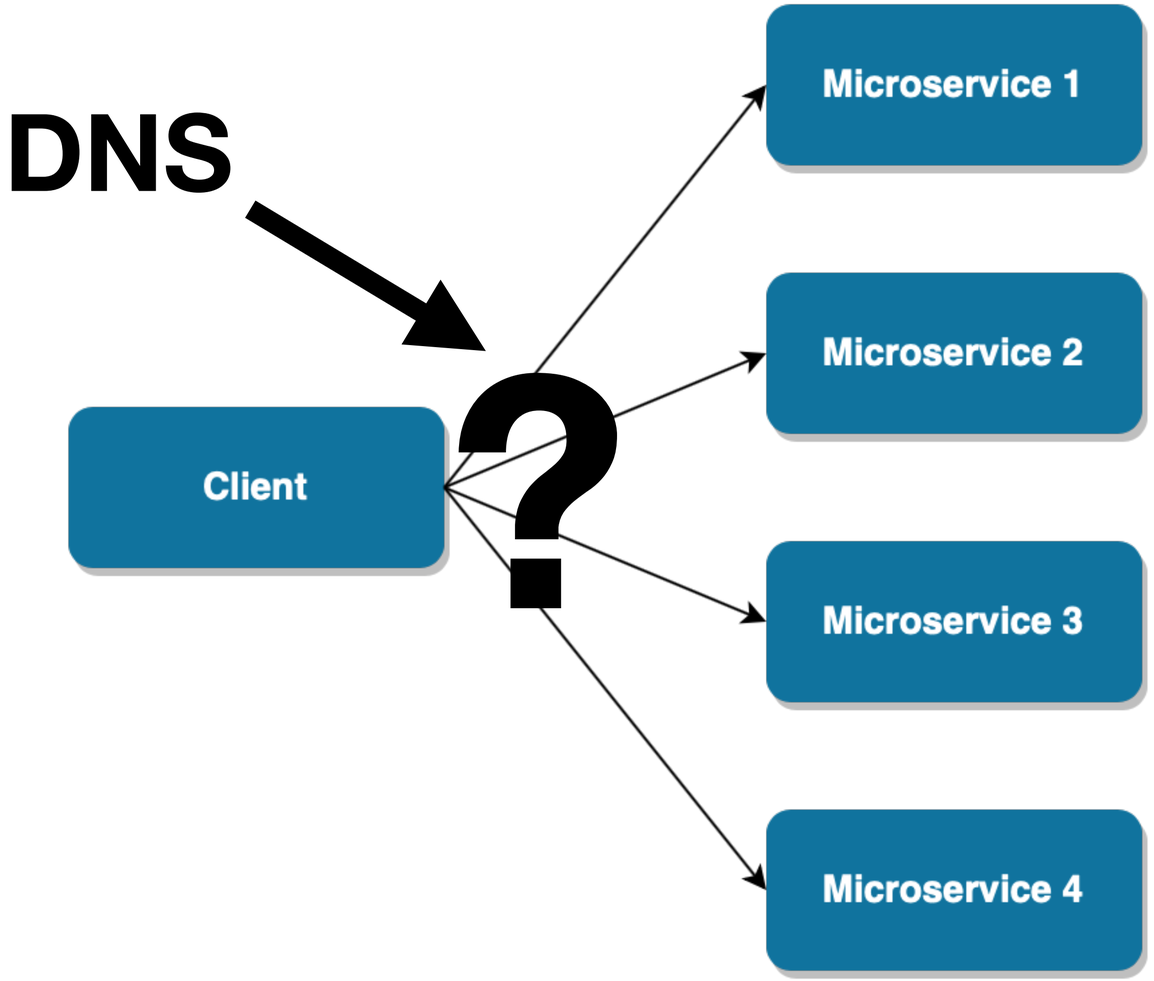
- ACL defined as single microservice
- Prevents duplication and redistribution
- Could be a Single Point of Failure

ACL on Gateway

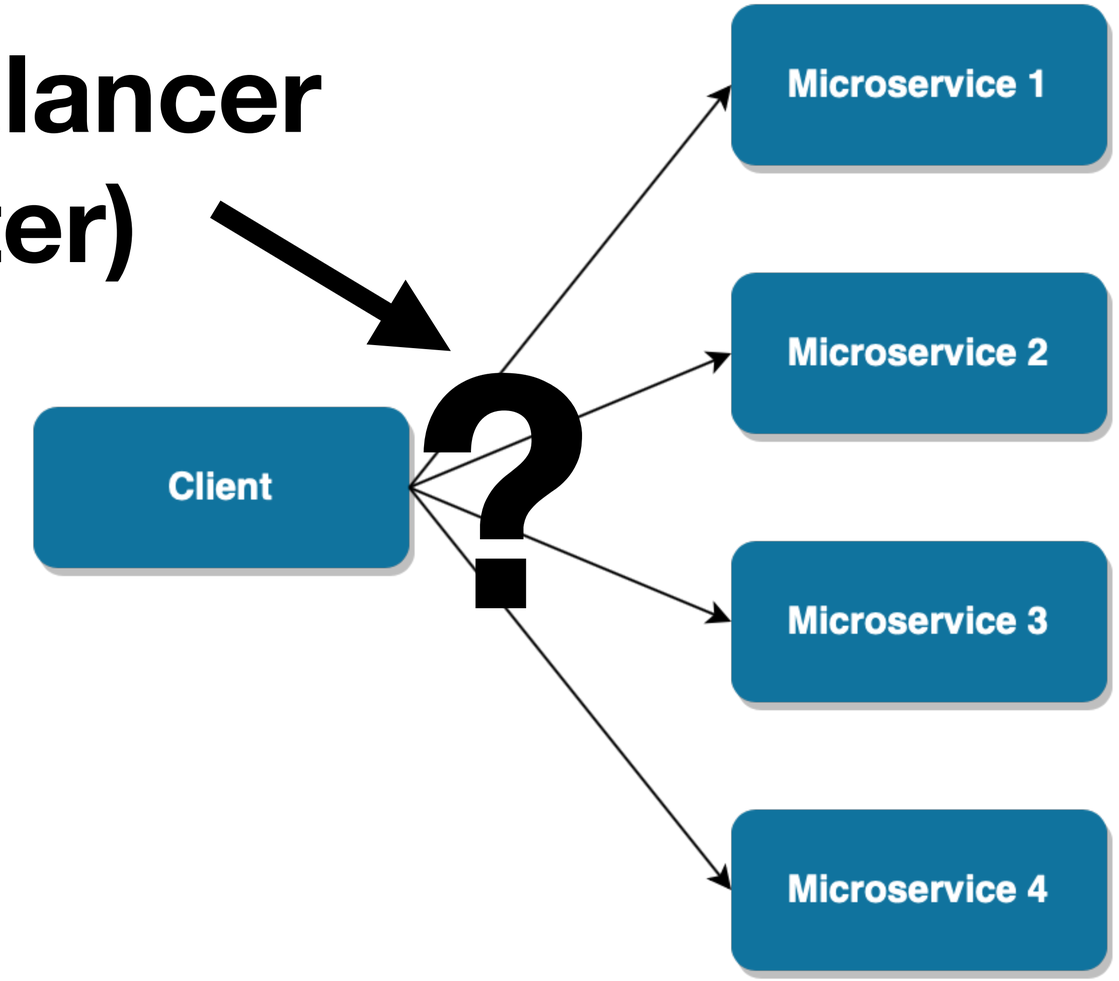
- ACL on Gateway (could be implemented as microservice)
- No duplicated ACL code in microservices
- No huge over-head, simplifies internal communication

Availability


Client
?
Microservice 1
Microservice 2
Microservice 3
Microservice 4

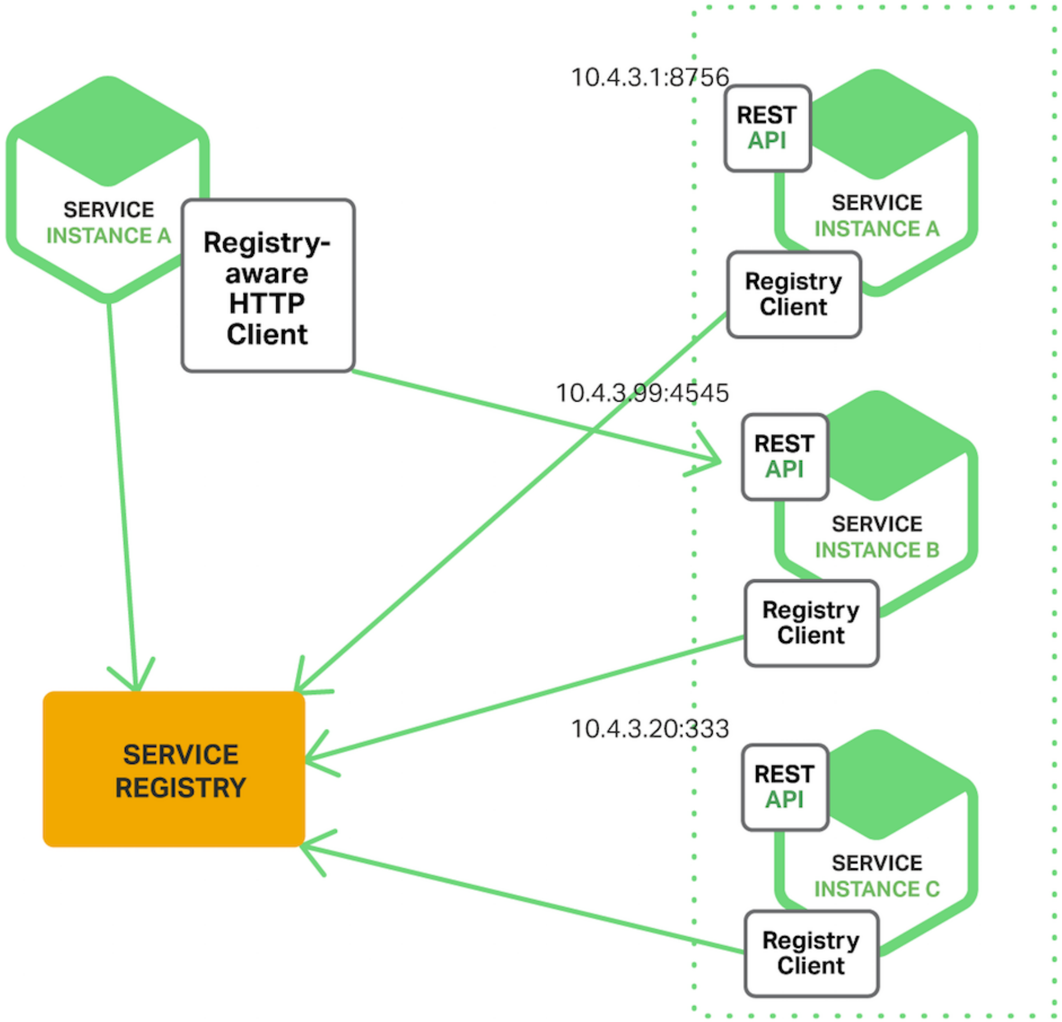

DNS
Client
?
Microservice 1
Microservice 2
Microservice 3
Microservice 4

Load Balancer (router)

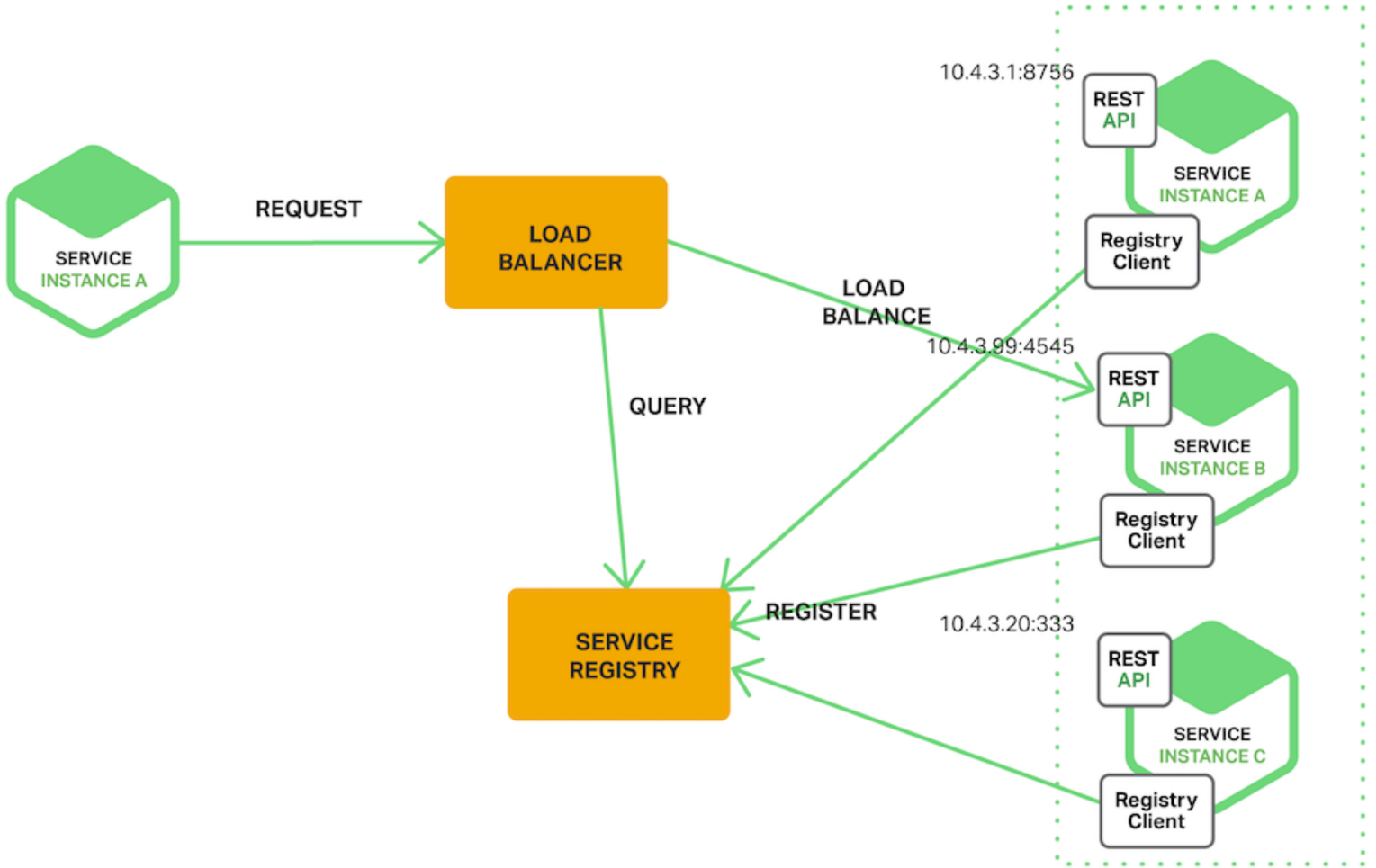


Service Discovery

Client-Side Service Discovery



Server-Side Service Discovery



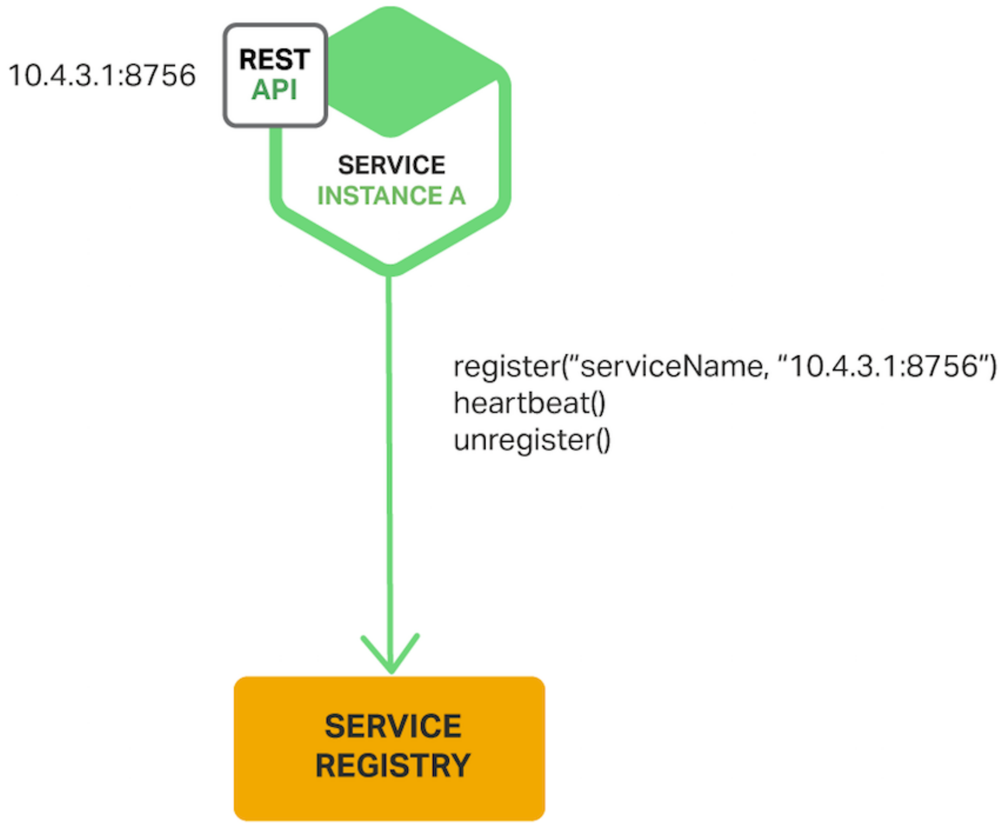
Service Registries

HashiCorp
Consul

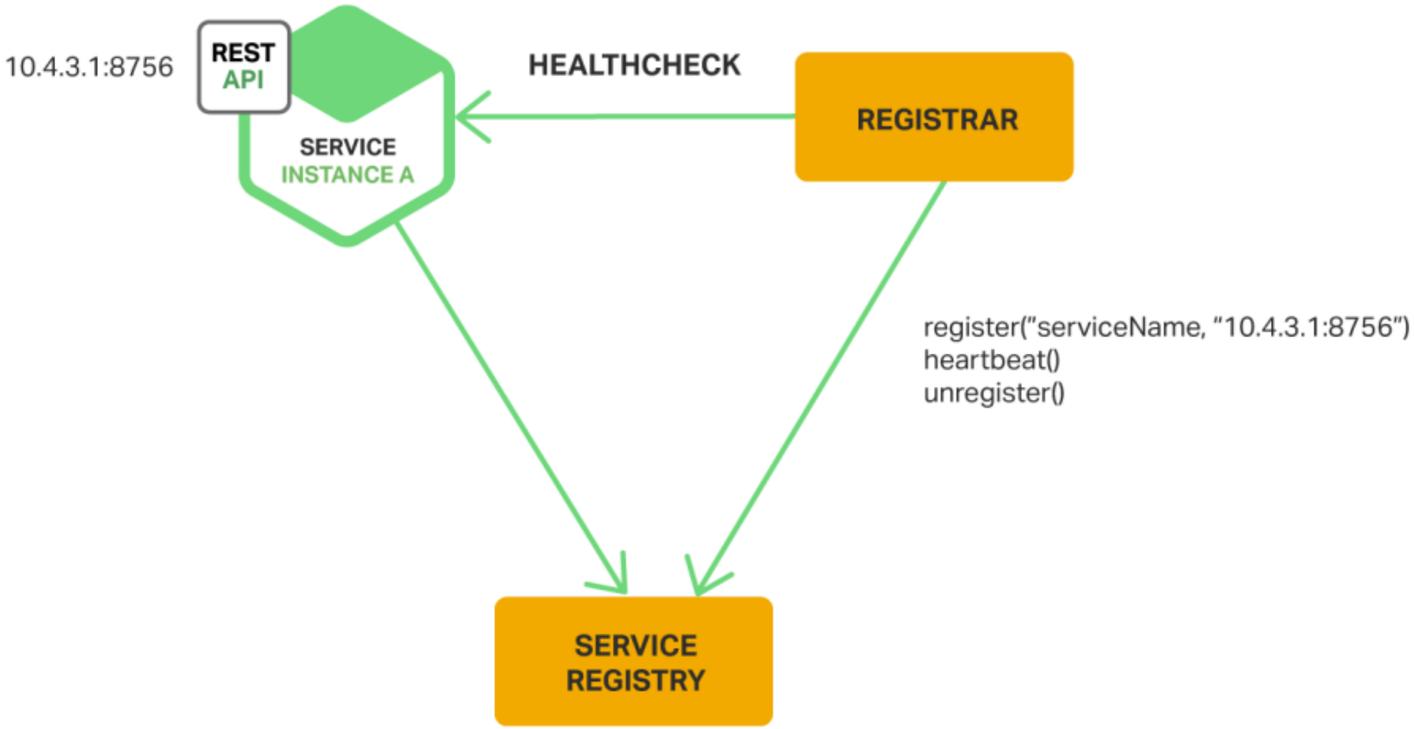
etcd

APACHE
ZooKeeper™

Self-Registration



3rd Party Registration



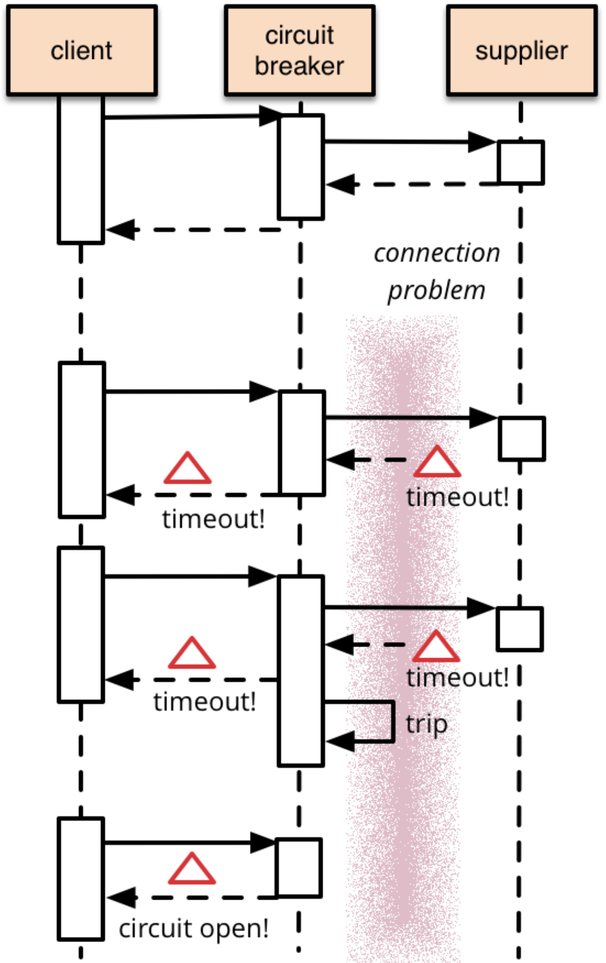
Consul

- Service Registry
- Healthchecking
- Key/Value Storage
- Secure Communication
- Multi Datacenter

Handle Failures

- Define network timeouts
- Limit no. of outstanding requests
- Provide fallbacks

Circuit Braker



Configuration

Externalized Configuration

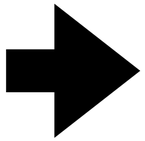
```
{
   "sectionA":{
      "valueA":"ABC123",
      "valueB":"XYZ321"
   }
}
```

One configuration to rule them all

```
{
   "sectionA":{
      "valueA":"ABC123",
      "valueB":"XYZ321"
   },
   "sectionB":{
      "valueC":"ABC123",
      "valueD":"XYZ321"
   }
}
```

One configuration to rule them all

```
{
   "sectionA":{
      "valueA":"ABC123",
      "valueB":"XYZ321"
   },
   "sectionB":{
      "valueC":"ABC123",
      "valueD":"XYZ321"
   }
}
```

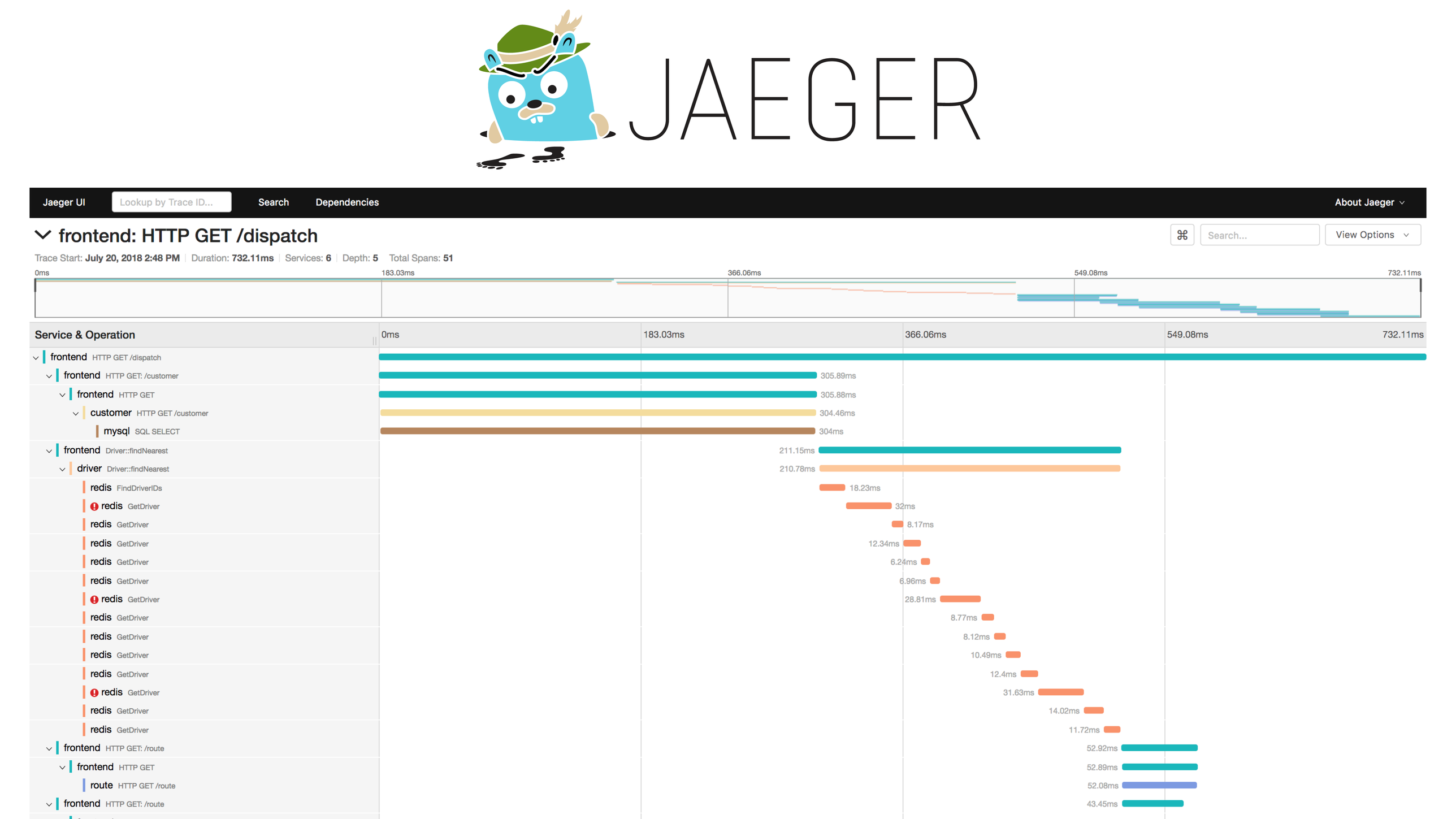


```
{
   "sectionA":{
      "valueA":"ABC123",
      "valueB":"XYZ321"
   }
}
```

(Logging)
Tracing

How to debug microservices?

- Aggregate all logs in one place
- Track Exceptions
- Trace requests
- Visualize


JAEGER
Jaeger UI
Lookup by Trace ID...
Search
Dependencies
About Jaeger
frontend: HTTP GET /dispatch
Search...
View Options
Trace Start: July 20, 2018 2:48 PM | Duration: 732.11ms | Services: 6 | Depth: 5 | Total Spans: 51
0ms
183.03ms
366.06ms
549.08ms
732.11ms
Service & Operation
frontend HTTP GET /dispatch
frontend HTTP GET: /customer
305.89ms
frontend HTTP GET
305.88ms
customer HTTP GET /customer
304.46ms
mysql SQL SELECT
304ms
frontend Driver::findNearest
211.15ms
driver Driver::findNearest
210.78ms
redis FindDriverIDs
18.23ms
redis GetDriver
32ms
redis GetDriver
8.17ms
redis GetDriver
12.34ms
redis GetDriver
6.24ms
redis GetDriver
6.96ms
redis GetDriver
28.81ms
redis GetDriver
8.77ms
redis GetDriver
8.12ms
redis GetDriver
10.49ms
redis GetDriver
12.4ms
redis GetDriver
31.63ms
redis GetDriver
14.02ms
redis GetDriver
11.72ms
frontend HTTP GET: /route
52.92ms
frontend HTTP GET
52.89ms
route HTTP GET /route
52.08ms
frontend HTTP GET: /route
43.45ms

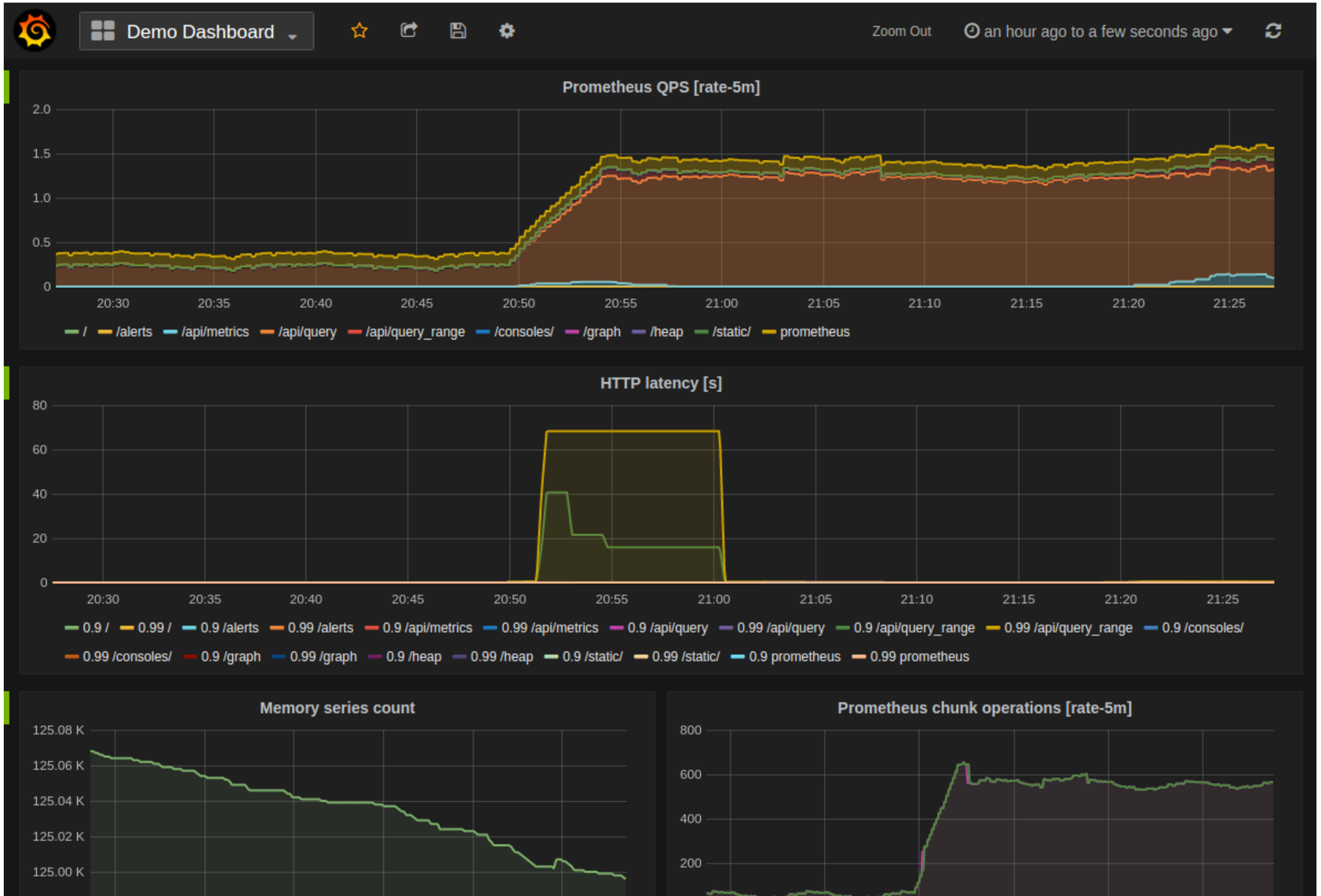
Measure

- Monitor relevant platform details:
  - execution time of db queries;
  - request execution time;
  - number of request per minute;
  - response times;
  - success/failure ratio;
  - resources usage

Prometheus


Prometheus Stats
Zoom Out
Last 5 minutes
Grafana Docs
Prometheus Docs
Uptime
1.5 hour
Local Storage Memory Series
37362
Internal Storage Queue Length
Empty
Prometheus
You're using Prometheus, an open-source systems monitoring and alerting toolkit originally built at SoundCloud. For more information, check out the Grafana and Prometheus projects.
Samples ingested (rate-5m)
7.310 K
7.305 K
7.300 K
7.295 K
7.290 K
7.285 K
7.280 K
11:37:00
11:37:30
11:38:00
11:38:30
11:39:00
11:39:30
11:40:00
11:40:30
11:41:00
11:41:30
prometheus
Samples Ingested
This graph displays the count of samples ingested by the Prometheus server, as measured over the last 5 minutes, per time series in the range vector. When troubleshooting an issue on IRC or Github, this is often the first stat requested by the Prometheus team.
Target Scrapes (last 5m)
4.200075
4.200050
Scrape Duration
5.005
5.004
Scrapes
Prometheus scrapes metrics from

Grafana


Demo Dashboard
Zoom Out
an hour ago to a few seconds ago
Prometheus QPS [rate-5m]
2.0
1.5
1.0
0.5
0
20:30
20:35
20:40
20:45
20:50
20:55
21:00
21:05
21:10
21:15
21:20
21:25
/
/alerts
/api/metrics
/api/query
/api/query_range
/consoles/
/graph
/heap
/static/
prometheus
HTTP latency [s]
80
60
40
20
0
0.9 /
0.99 /
0.9 /alerts
0.99 /alerts
0.9 /api/metrics
0.99 /api/metrics
0.9 /api/query
0.99 /api/query
0.9 /api/query_range
0.99 /api/query_range
0.9 /consoles/
0.99 /consoles/
0.9 /graph
0.99 /graph
0.9 /heap
0.99 /heap
0.9 /static/
0.99 /static/
0.9 prometheus
0.99 prometheus
Memory series count
125.08 K
125.06 K
125.04 K
125.02 K
125.00 K
Prometheus chunk operations [rate-5m]
800
600
400
200

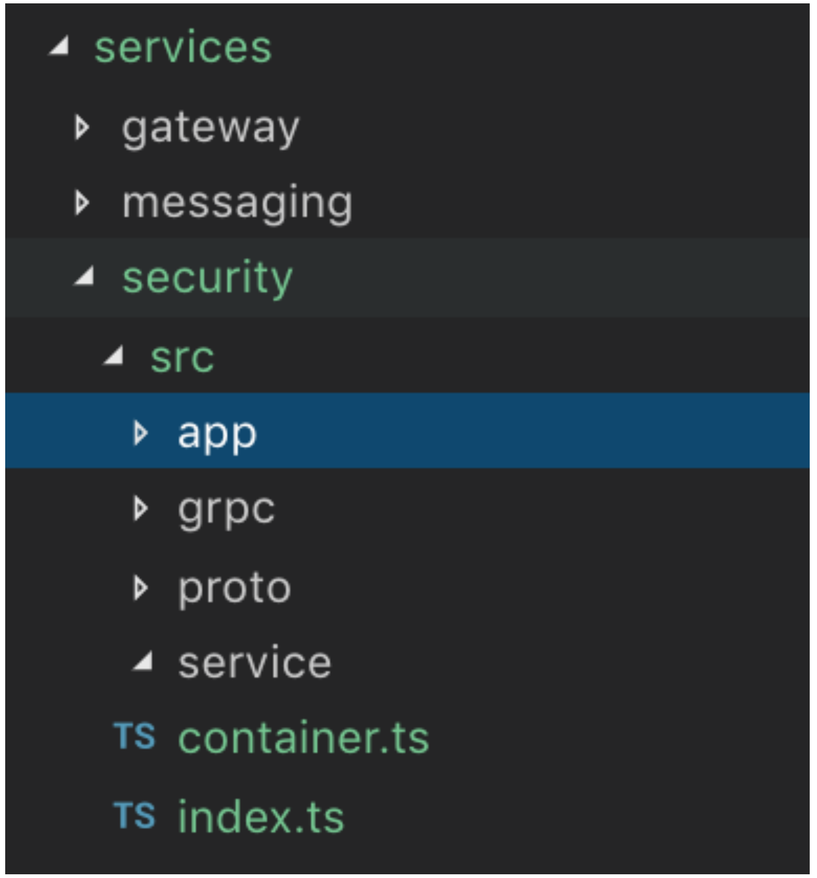
Now, we can do microservices the right way

Can we do them fast?

Define standards / contracts

Embrace tooling

Example: Service structure generation



Example: GRPC code generation

```
message Message {
  string content = 1;
  string username = 2;
}
```

```
export type Message = {
  content: string;
  username: string;
};
```

Example: GRPC code generation

```
service MessagingService {
    rpc sendMessage (SendMessageRequest) returns (SendMessageResponse) {};
    rpc getLatestMessages (GetLatestMessagesRequest) returns (GetLatestMessagesResponse) {};
    rpc joinChat (JoinChatRequest) returns (stream Message) {};
}
```

```
export interface MessagingServiceHandlers {
  sendMessage: (request: SendMessageRequest) => Promise<Response<SendMessageResponse>>;
  getLatestMessages: (request: GetLatestMessagesRequest) => Promise<Response<GetLatestMessagesResponse>>;
  joinChat: (request: JoinChatRequest) => Promise<Response<Message>>;
}
```

How to organize microservices in code repositories?

Multi-repository

- Pros:
  - better owner (team) separation;
  - clear scope

- Cons:
  - hard to keep consistency (versioning is complex);
  - risk of becoming a monolith;
  - introduces more CI/CD setup costs

Mono-repository

- Pros:
  - easy to maintain consistency;
  - easier versioning
- Cons:
  - different teams may affect each other;
  - might lead to tight coupling;
  - longer build times

What we didn’t covered today

- Databases
- Testing
- CI / CD as essential tool
- Destructive Testing
- Deployments
- many, many more…

youtu.be/D8I19VEokBA

Summary

- **think twice** if you really need a microservice architecture
- ensure you have all necessary resources to operate distributed systems
- make cold and educated decisions what's **best for your case**, not netflix
- embrace contracts and tooling

Resources

- microservices.io
- martinfowler.com/microservices
- scs-architecture.org
- codingthearchitecture.com
- nginx.com/blog/introduction-to-microservices

Don’t miss our upcoming webinars!
Sign up for our newsletter! ✍️

tsh.io/newsletter

Thanks 😉

tsh.io