Microservices with Java and Spring Boot

Jan Ypma

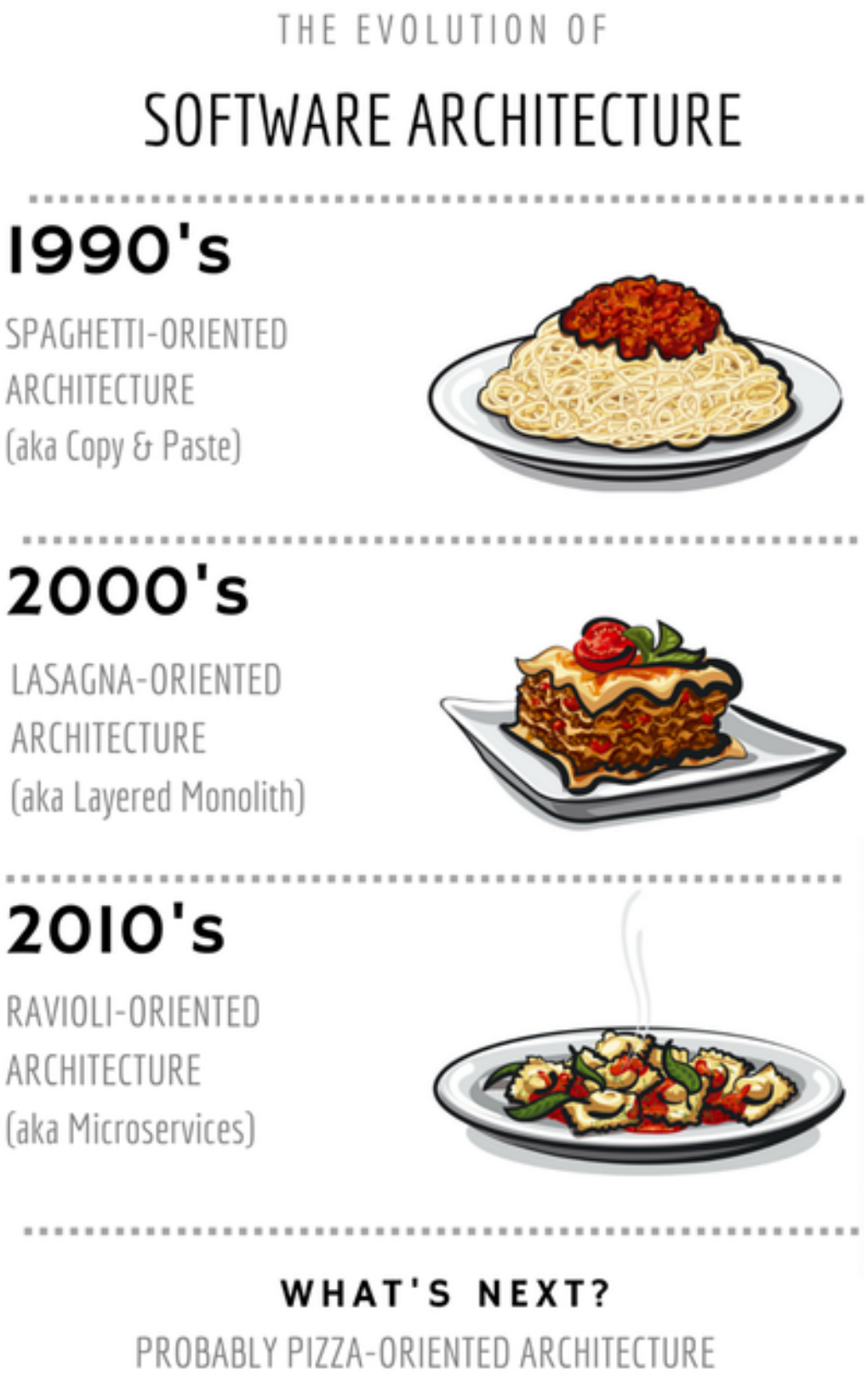
November 9, 2022

Contents

1 Introduction

1.1 Getting started

1.1.1 Welcome



1.1.2 This course

Freely available here: https://github.com/jypma/LB3212-services

- PDF of slides will be available at completion
- Pull requests with fixes to our typos are welcome :-)

1.1.3 About your instructors

- Jan Ypma
  - Java, Scala, Groovy, C++, Rust, Lisp
  - Contributor to various open source projects
  - Fan of functional programming and distributed systems
  - Agile coach
  - `jan@ypmania.net`, https://linkedin.com/in/jypma
- Jakob Bendsen
  - 20+ years of IT and Java experience
  - Teaching at ITU and numerous courses in Danish IT industry
  - Experience with Java/Jakarta EE and Spring (Boot)
  - Likes Kotlin and elegant programs
  - `jakob@logb.dk`, https://linkedin.com/in/jbendsen

1.1.4 Schedule, day 1

Time	Duration	Activity	Weight
09:00	00:10	Welcome, Outline/Agenda	
09:10	00:20	Round table introductions	
09:30	00:15	What's a microservice	
09:45	00:15	The 8 fallacies of distributed computing	
10:00	00:30	Design for resilience	
10:30	00:20	Break(out) 1	
10:50	00:20	Discussion of breakout results	
11:10	00:60	Infrastructure architecture	
12:10	00:20	Break(out) 2	
12:30	00:30	Lunch	
13:00	00:20	Discussion of breakout results	
13:20	00:40	Data architecture	
14:00	00:40	Event storming: Events	
14:40	00:10	Break	
14:50	00:30	Event storming: Commands and Actors	
15:20	00:20	Event storming: Aggregates	
15:40	00:20	Wrap-up, reserved time for extra subjects	

1.1.5 Talk about yourself

- My Background
- What do I hope to get out of the course
- What's the smallest service I've ever written

1.2 What's a microservice

1.2.1 Definition

- *Service*
  - One operating system process (often on its own server)
  - Exposes an API (sometimes also a UI)
- *Micro*
  - Theory: It's small
  - Practice: There are many
  - Independently deployable

1.2.2 Philosophy

- Business needs evolve
- Team composition changes
- Services should be disposable (design to be replaceable)
  - Rebuilt in 1-3 months
- Per service, use best technology matching experience and requirements

1.2.3 Service scope

- Service belongs to one team
  - Team is responsible for entire service software life cycle
- Data store belongs to one service
- Independently deployable

1.2.4 Use cases

- Embrace Conway's law: One system belongs to at most one team
- Monoliths are fine to start with
  - Time to market and technical debt vs. holistic design
- Strangler pattern

1.3 The 8 fallacies of distributed computing

1.3.1 Which of these is true?

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

1.3.2 How well does the following abstraction help?

```
MyResponseType callMyRemoteService(String command, byte[] data)
```

2 Design for resilience

2.1 Service failure

2.1.1 Your (or your colleague's) service will be down

P(everything working) = P(one service is working) ^ $n_{services}$

Our service is up 99% of the time!

Well, we have about 30 microservices, each with 3 copies. That means that 63% of the time, at least one service is down somewhere.

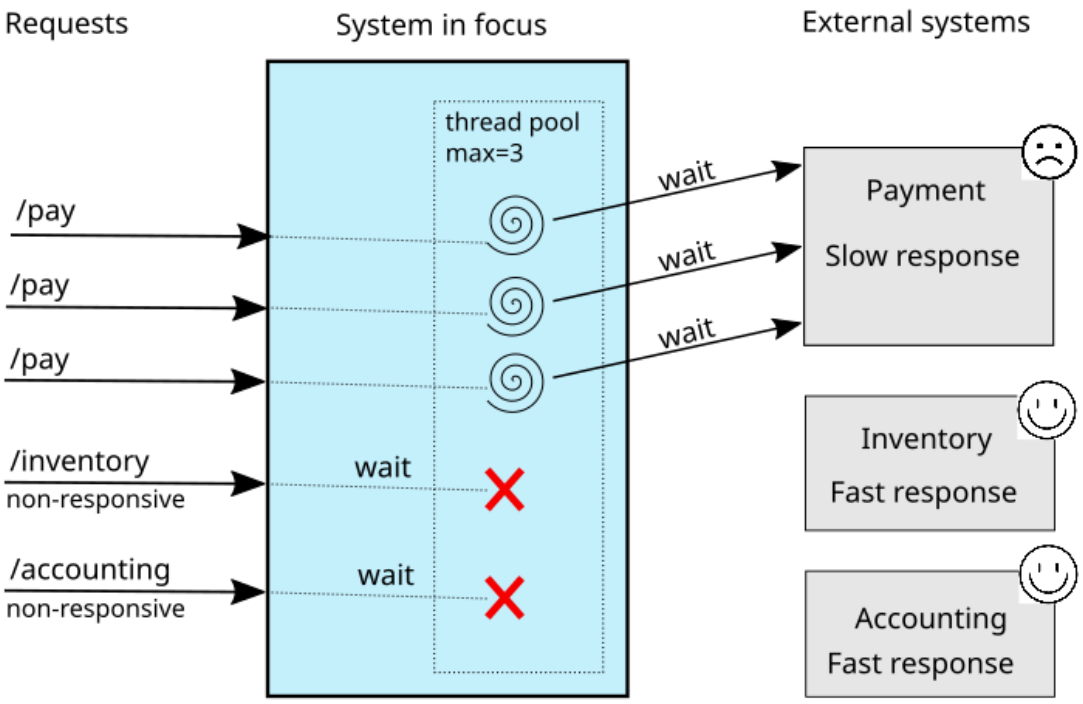
2.2 Creating services

2.2.1 Guidelines

- Prefer sharded (partitioned) data stores over single points of failure
- Idempotency for all incoming data
- Always deploy more than 1 copy
  - Investigate the need for a cluster-aware distributed framework
- Have a *Service dashboard* with metrics (more on that later)
- Use `Bulkhead` to protect finite resources

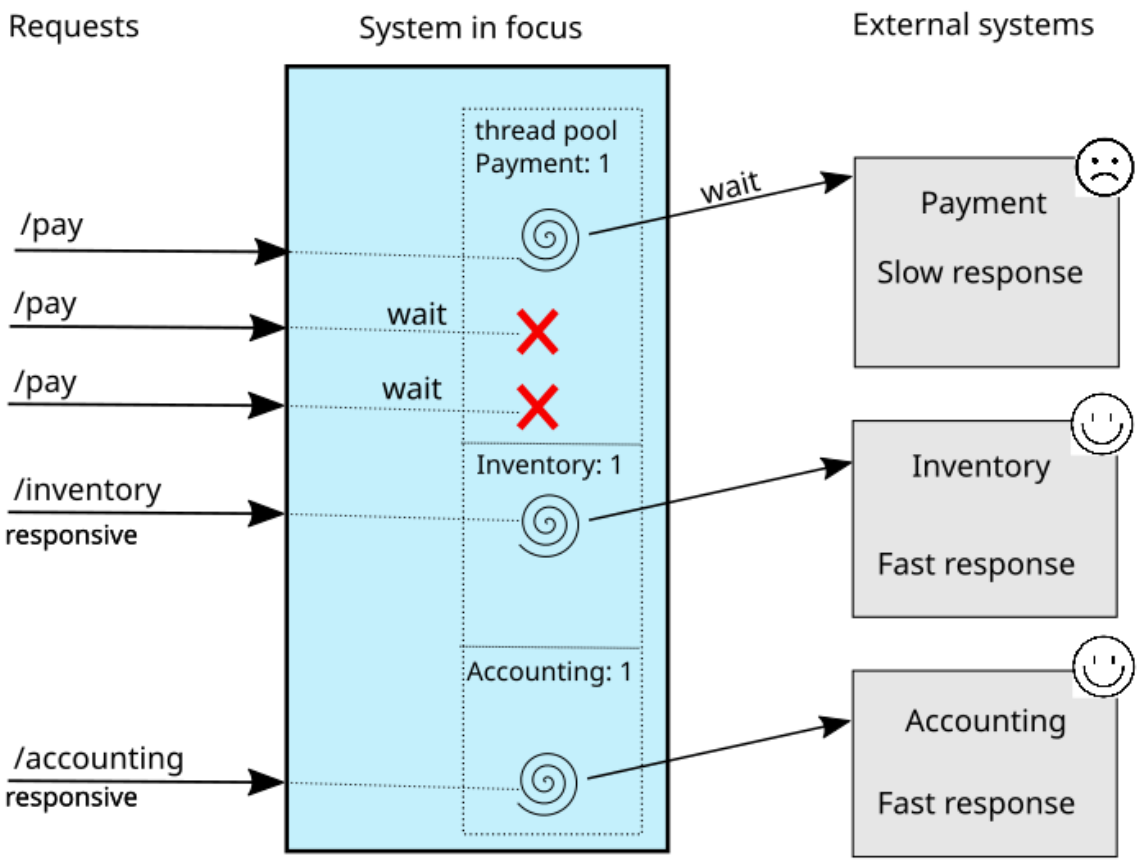
2.2.2 Bulkhead

A single resource pool is covering multiple types of application requests



2.2.3 Bulkhead in place

- Semaphore with an optional timed queue in front
- Other parts of the resource pool are still accessible



2.2.4 Measure service quality

- Service Level Objective (SLO)
  - Metric that indicates a healthy service to you, e.g.
    - *"The 99th percentile of HTTP response times is at most 300ms"*
    - *"At least 99.9% of HTTP requests result in a successful response"*
  - Typically only internally measured and/or agreed between teams
- Service Level Indicator (SLI)
  - An actual number that indicates the current value of an *SLO*, e.g.
    - 99th percentile response time
    - 24-hour window success rate of HTTP requests
- Service Level Agreement (SLA)
  - Part of a contractual obligation (sometimes legally binding) between parties
    - *"The 95th percentile of HTTP response times is at most 1000ms"*
    - *"At least 99% of HTTP requests result in a successful response"*
  - Typically results in a stricter SLO being applied internally

2.3 Consuming services

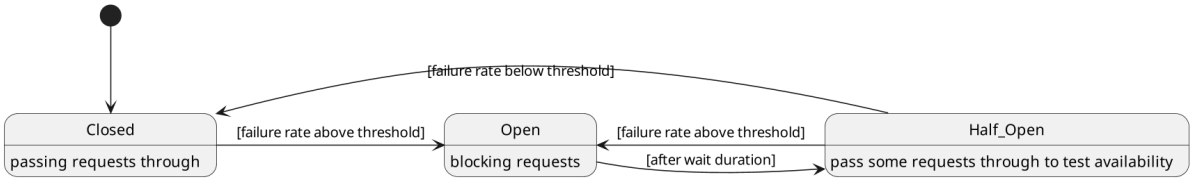
2.3.1 Guidelines

- Design for failure
  - Have methods/functions reflect doing I/O

  - Make time (and timeouts) explicit
  - Use `Circuit Breaker` where applicable
- Fail fast
  - `System.exit(1)` is a viable error handler

2.3.2 Circuit breaker

- Smart state machine towards 1 backend
  - *Closed*: Everything is working normally
  - *Open*: We've determined that the backend is down, and block requests
  - *Half-open*: We're allowing a few requests through, to test the waters



2.4 Guidelines

2.4.1 Microservice pitfalls

- Service co-dependencies
  - Keep HTTP calls one way only
  - Plugin pattern
- Nested synchronous service calls
  - Added latency and failure possibility
  - Avoid these with event sourcing
  - Replicate data instead, or call asynchronously when possible

2.4.2 Need more inspiration?

- The twelve-factor app, https://12factor.net/
- Provides sensible suggestions on a lot of topics
  - Port binding, dev/prod differences, admin processes
- Not the only way (geared towards ruby/python), but worth a thorough read

3 Break(out) 1

3.1 Service discovery

3.1.1 Introduction

- Break into teams of 2-4 people (20 min)
- Discuss the services and projects you've been a part of (here or at a previous employer), and identify:
  - Examples of a microservice
  - Examples of *definitely NOT* a microservice
- For each service found, describe how *resilient* the given service was
  - Usage or absence of *bulkhead* and/or *circuit breaker*
  - Usage or absence of clustering / replication

3.1.2 Discussion

- Describe the services you have found

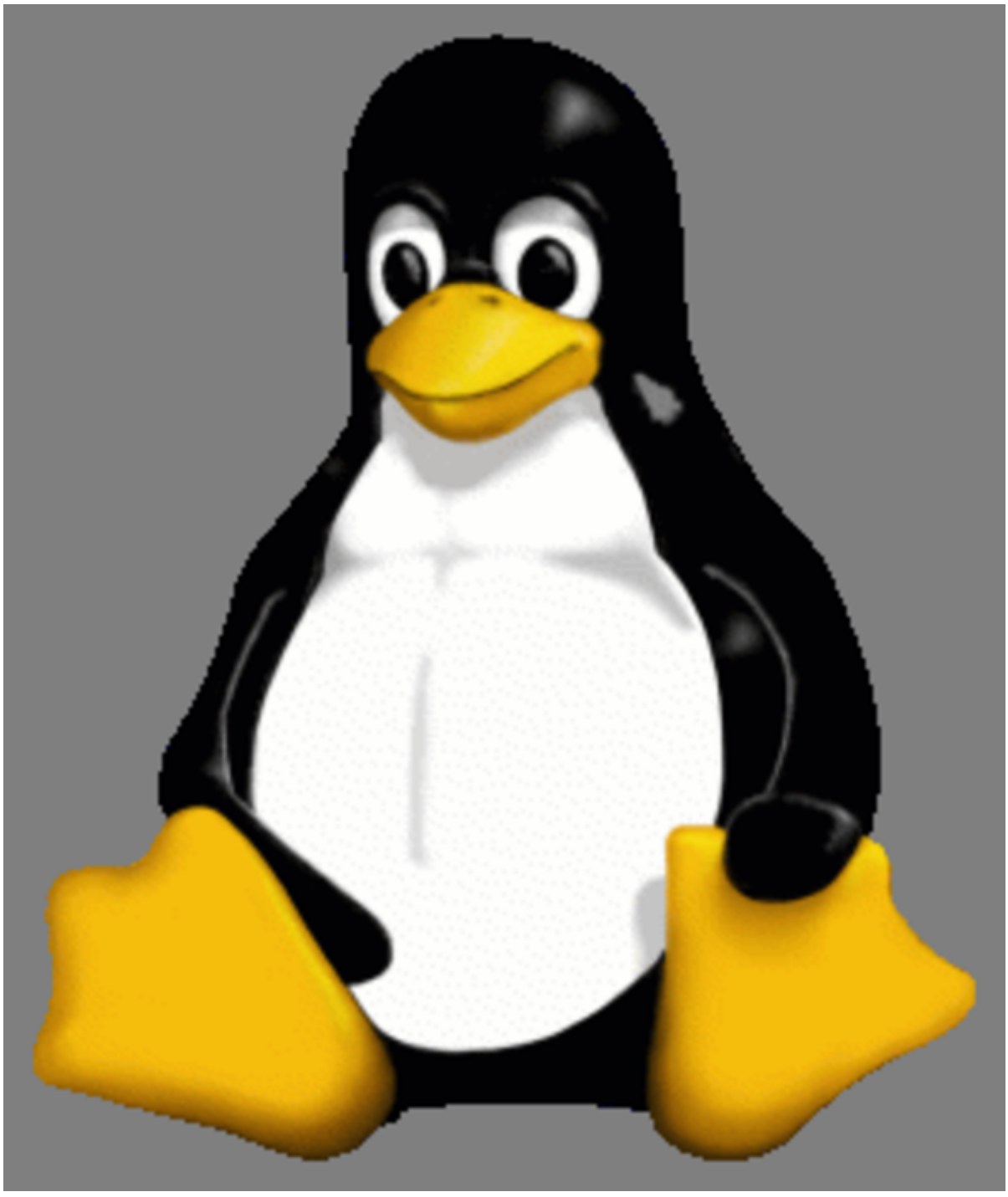
4 Infrastructure architecture

4.1 It's a linux world

4.1.1 About linux

4.1.2 Get familiar with linux

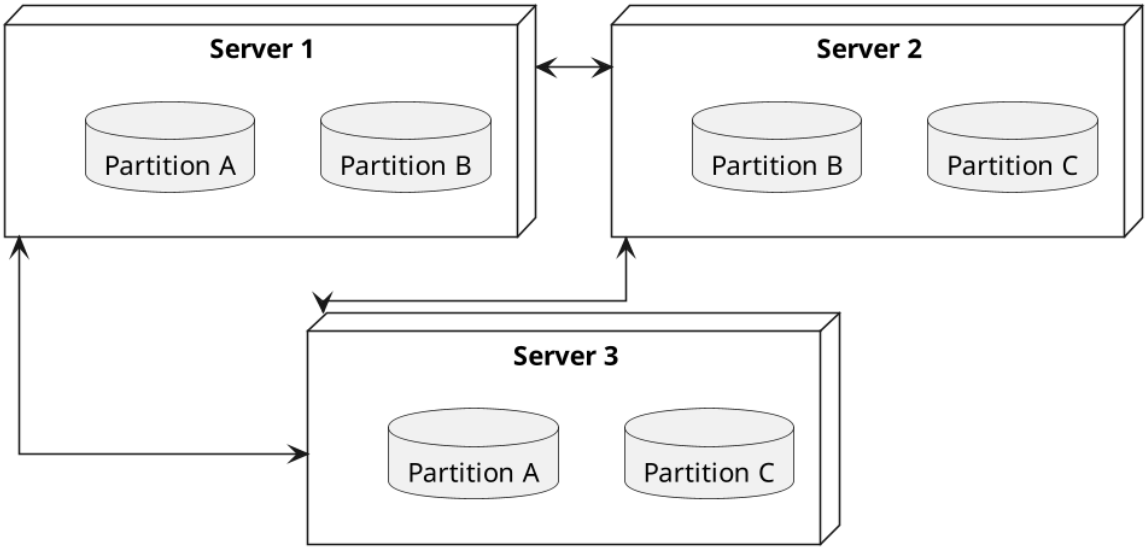
- Micro services are a linux world
- It's easier than ever to get started
  - WSL 2 (some integration, less "linux", and has issues)
  - VirtualBox with e.g. Ubuntu (real linux)
  - Dual boot e.g. Ubuntu
  - Just get a Raspberry Pi



4.2 Partitioned data stores

4.2.1 Partitioned data stores: introduction

- All data is split into partitions (also called *shards*), which are copied onto servers
- For each data element, a *key* determines which partition it's stored on



4.2.2 Partitioned row stores

Each *row* has a *key* that specifies which partition(s) store data for that row. Data is typically stored in columns, following a schema.

- Open source: Cassandra
- Amazon: DynamoDB, Keyspaces
- Google: BigTable
- Azure: Cosmos DB (with Cassandra API)

4.2.3 Example cassandra queries

- Creating a table

```
CREATE TABLE chat_messages (
  roomId int,
  seqNr int,
  edited timestamp,
  userId int,
  message text,

  PRIMARY KEY (roomId, seqNr)
);
```

- Table must have a primary key
- Part of the primary key is the *partition* key, which dictates how the data is partitioned (sharded)
- Inserting (or updating) rows

```
INSERT INTO chat_messages (roomId, seqNr, edited, userId, message)
  VALUES (1, 1, NOW(), 42, 'This is my message');
```

- This will insert (or overwrite) the row for the data's primary key values
- `UPDATE` also exists, and has the same semantics
- Did somebody say this is NoSQL?

4.2.4 Partitioned queues

Messages sent to a queue (sometimes called topic) are distributed to partitions, based on a *key*. Messages typically small (some services have upper limit of 64kB).

- Open source: Kafka
- Amazon: SQS
- Google: Cloud Pub/Sub
- Azure: Storage Queue (*) , Service Bus (*)

(*) *not partitioned, size-limited*

4.2.5 Partitioned search

Full-text search is often important when dealing with data.

- Open source: Elasticsearch, SoLR
- Amazon: Hosted elasticsearch
- Google: Hosted elasticsearch
- Azure: Hosted elasticsearch

4.3 Single-server data stores

4.3.1 Single-server data stores: introduction

- Many moving parts needed to make primary/replica failover work
  - PostgreSQL: Multiple servers possible, but failures leak to the client. `pgBouncer` as alternative.
  - MariaDB: Multiple servers possible with failover, fail-back is a manual process
  - RabbitMQ: Multiple servers possible with failover, but fail-back doesn't work in Spring (AMQP-318)
- If you choose these, make failover testing part of your CI

4.3.2 PostgreSQL

- Relational database with a strong history of transactional correctness
- Very high performance
- Modern features
  - Native JSON support with indexes
  - Add indexes without locking tables
- Single-server, but flexible native replication options
  - Multiple read replicas
  - Subset-read replicas (*"logical replication"*)
- Database-level sharding software exists, but application-level sharding is recommended

4.3.3 RabbitMQ

- Message queue with focus on performance
- Original architecture single-server
  - Later extended with *Mirror Queues* (primary/replica)
  - Extended with *Quorum Queues* in 2019 (raft)
    - No message TTL, no message priorities
    - All cluster members have all data
    - All messages in memory! (in addition to storage)

4.3.4 RabbitMQ Data consistency

- AMQP "transaction"
  - Covers only a single queue
  - "Slow" (fsync for every transaction)
- *Publisher confirms*
  - Asynchronous message from RabbitMQ to client (after fsync): `basic.ack` or `basic.nack`
  - Impossible to predictably deal with lost broker connection (risk duplicate, risk lost messages)
- Manual *Consumer acknowledgement*
  - Consumer sends message to RabbitMQ to confirm handling of message is complete
  - `basic.ack`, `basic.nack(requeue)`, `basic.nack(no requeue)`
  - This is async, so no guarantee that the server receives it
    - Two generals agree

4.4 Monitoring and alerting

4.4.1 Introduction

- Logging need not be a cross-cutting concern
  - Create monitored metrics instead
- Your service dashboard is as important as your public API
  - Have metrics on *everything*
  - Dashboard should be visible to and understandable by non-team members
- Be aware of your resource usage, check all environments at least daily

4.4.2 Protocol variations

- Push-based (`statsd`)
  - Application periodically (10 seconds) sends UDP packet(s) with metrics
  - Simple text-based wire format
  - Composes well if running with multiple metrics backends
  - Advantages: composability, easy to route, less moving parts
- Pull-based (prometheus)
  - Database calls into microservice periodically (10 seconds) over HTTP
  - Service needs to run extra HTTP server
  - Does not compose (multiple metrics backends need to be known on the prometheus side)
  - Advantages: less timing-sensitive

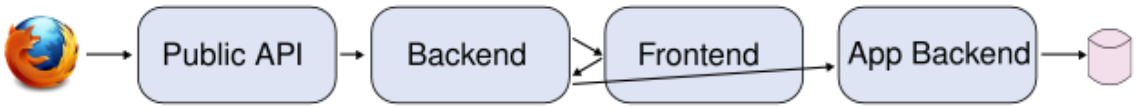
4.4.3 Metrics terminology

- Different frameworks use different terms
- Micrometer uses the following:
  - *Counter* (sometimes called *event*): An occurrence of a discrete event
    - e.g. a request coming in
  - *Gauge*: The size of a single measurable quantity (and its unit)
    - e.g. the number of active TCP connections
  - *Timer*: The duration of an activity
    - e.g. the response time to a request
  - *Distribution summary* (sometimes called *histogram* or even *gauge*): Recorded values (and units) that go with events
    - e.g. the size of incoming requests in bytes

4.5 Request tracing

4.5.1 Complex service dependencies

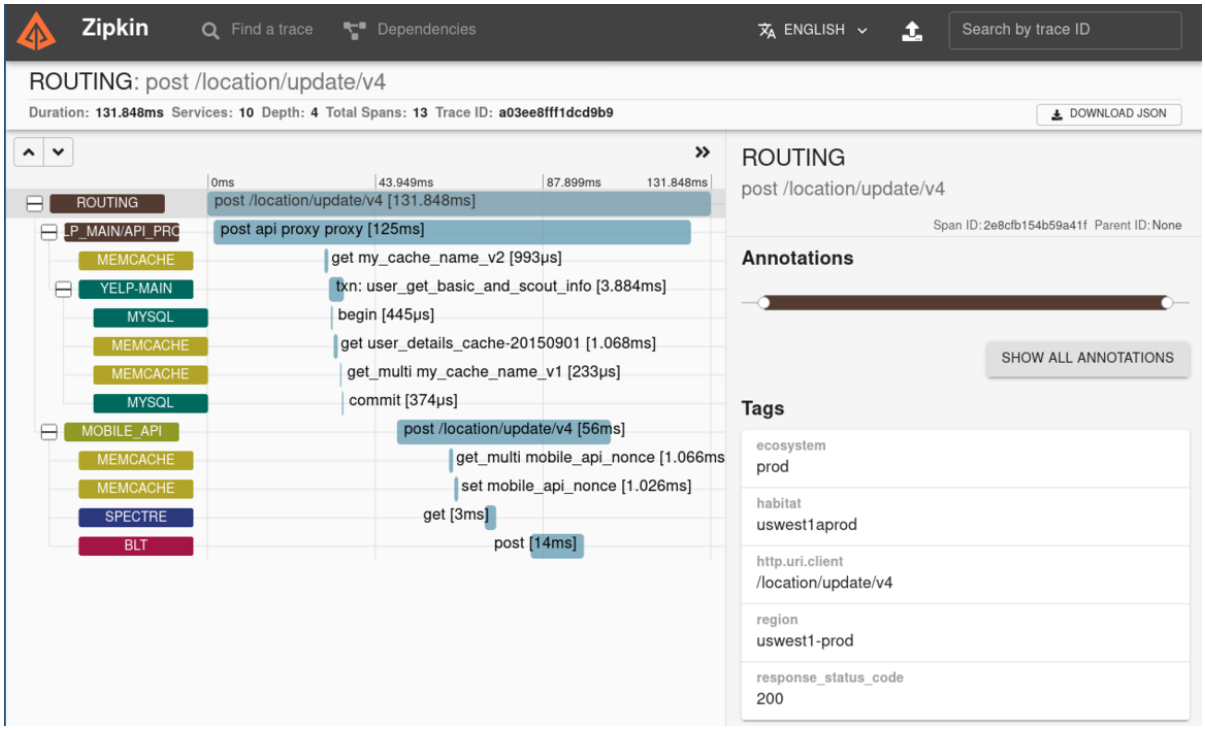
- Services can have complex calling stacks



- When something breaks, it's good to have a trace
- Other reasons
  - Identify performance problems
  - Find bottlenecks
  - Track resource usage

4.5.2 Two mature solutions

- Jaeger and Zipkin
  - Both have vast library and framework support
  - Many metrics framework support both backends



4.6 Deployment

4.6.1 Hosted, semi-hosted or self-hosted?

- Learning a new data store technology
  - Reliability guarantees
  - Scalability and performance characteristics
  - API
  - Installation and operation (for developers)
  - Installation and operation (in production)
- You can save on the last bullet, but not on the others
- Self-hosted
  - You install and run everything yourself
    - Kafka, Cassandra, Elasticsearch
    - Typically on Docker & Kubernetes
  - Can re-use knowledge and code between development and production
- Semi-hosted
  - Cloud provider installs and operates existing (typically open source) software for you
  - But you still have to pick server size and count
  - You're billed per server
- Hosted
  - Cloud provider installs and operates everything for you
  - You're billed per logical storage unit (e.g. database row or queue message)

4.6.2 Pets vs. cattle

- *Pets*: Traditional server management
  - Servers have cute names
  - Some server names I've seen: `pinkie`, `oink`, `tardis`, `deepthought`, `zeus`
  - Everyone know the peculiarities of each server
- *Cattle*: Cloud server management
  - Servers have only a logical ID or number
  - Hardware setup, rack and/or location
  - Find an available server to put your service on

4.6.3 Virtualization and containerization

- First, there was plain hardware
- VM abstraction
  - Decoupling of multiple roles of one server
  - Memory and disk overhead
  - Linux optimizations (kernel shared memory)
- Linux can do many of this natively
  - *Namespaces*: Hide processes from each other
  - *Cgroups*: Limit resource usage
- Containers to make it fast and efficient
  - VM: GBs
  - Docker (ubuntu): 100's of MB
  - Docker (alpine): MBs
  - Instant startup

4.6.4 Docker

- Limited to linux in this course
- Lightweight layer over native cgroups isolation
- Dockerfile

```
FROM node:12-alpine
RUN apk add --no-cache python g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

- Layers
- Volumes
  - Handling of persistent data
- Port mapping
- User mapping
- Don't run as root

4.6.5 Docker-compose

```yaml
version: '3.1'

services:

  db:
    image: postgres:13.2-alpine
    # Uncomment this to have the DB come up when you start docker / your laptop:
    #restart: always
    environment:
      POSTGRES_USER: demo
      POSTGRES_DB: demo
      POSTGRES_PASSWORD: example
    ports:
      - 5432:5432

  rabbitmq:
    image: rabbitmq:3.8.16-alpine
    # Uncomment this to have the DB come up when you start docker / your laptop:
    #restart: always
    ports:
      - 5672:5672    # AMQP
      - 15672:15672  # Web UI
```

- Groups several docker containers and storage
- Ideal for local testing

4.6.6 Kubernetes

- Manages a cluster of distributed docker containers with copies
  - *Pod*: Combination of one or more docker containers and their configuration
  - *Configmap*: Extra settings for pods, typically becoming a volume in the pod
  - *Deployment*: Automatic replicas and distributed upgrades for pods (and other resources)
- Ideal for production
- Configure Memory requests and limits
- Configure CPU requests
- Get comfortable getting thread and heap dumps
- Heap dump on out of memory (this *will* happen)
  - `-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/dumps` to an `emptyDir` volume

4.7 Configuration

4.7.1 Handling of externalized values

- Externalize "magic numbers" and strings
- Embrace your framework's ability to have *internal* and *external* configuration
  - *Internal* (inside docker container) has defaults and values that don't really change
  - *External* (mounted as a volume) has settings specific for that environment and/or server
- Changes to configuration files

  - Kubernetes: Configmap change does *not* restart the pod
  - Hot reloading? Not in spring boot (watch file and shutdown instead)
- Environment variables for secrets: don't do it (leaking to docker, monitoring tools)
  - use files instead
- Environment variables for service injection: don't do it (ordering issues)
  - use dns instead (e.g. dns-java, akka discovery, [...])

4.8 Load balancer

4.8.1 Allowing the world to call your service

- Deployed kubernetes services only reachable within the cluster
- Need to define an `ingress`
  - HTTP-level (NGinx) or TCP-level (HAProxy)
  - Provided by your native cloud provider
  - Hybrid setups
- Additional, external, load balancer in front of ingress controller

5 Break(out) 2

5.1 Infrastructure discovery

5.1.1 Introduction

- Resume in your teams
- Which pieces of infrastructure exist around the services you discovered?
- Who "owns" or maintains them?
- How can you set up new infrastructure?
- Look at all categories of infrastructure:
  - Servers
  - Data stores
  - Load balancers and gateways
  - Monitoring and dashboards
  - Others

5.1.2 Discussion

- Describe the infrastructure you have found

6 Data architecture

6.1 Domain-driven design

6.1.1 Introduction

- Software methodology
  - *Names in code must names used by the business*
- Popularized in 2003 by Eric Evans in his book
- Simple guideline lead to extremely useful patterns

6.1.2 Bounded context

- Reasoning about complex business processes requires abstractions
  - A *domain model* implements these abstractions as code
- Abstractions, and hence models, have a limited applicability
- *Bounded context* makes this explicit
  - When creating a domain model, evaluate the scope of your design
  - Create sub-domains when you encounter them
  - Describe the bounds for your domain
- Bounded context is often a good candidate for Microservice boundaries

6.1.3 Ubiquitous language

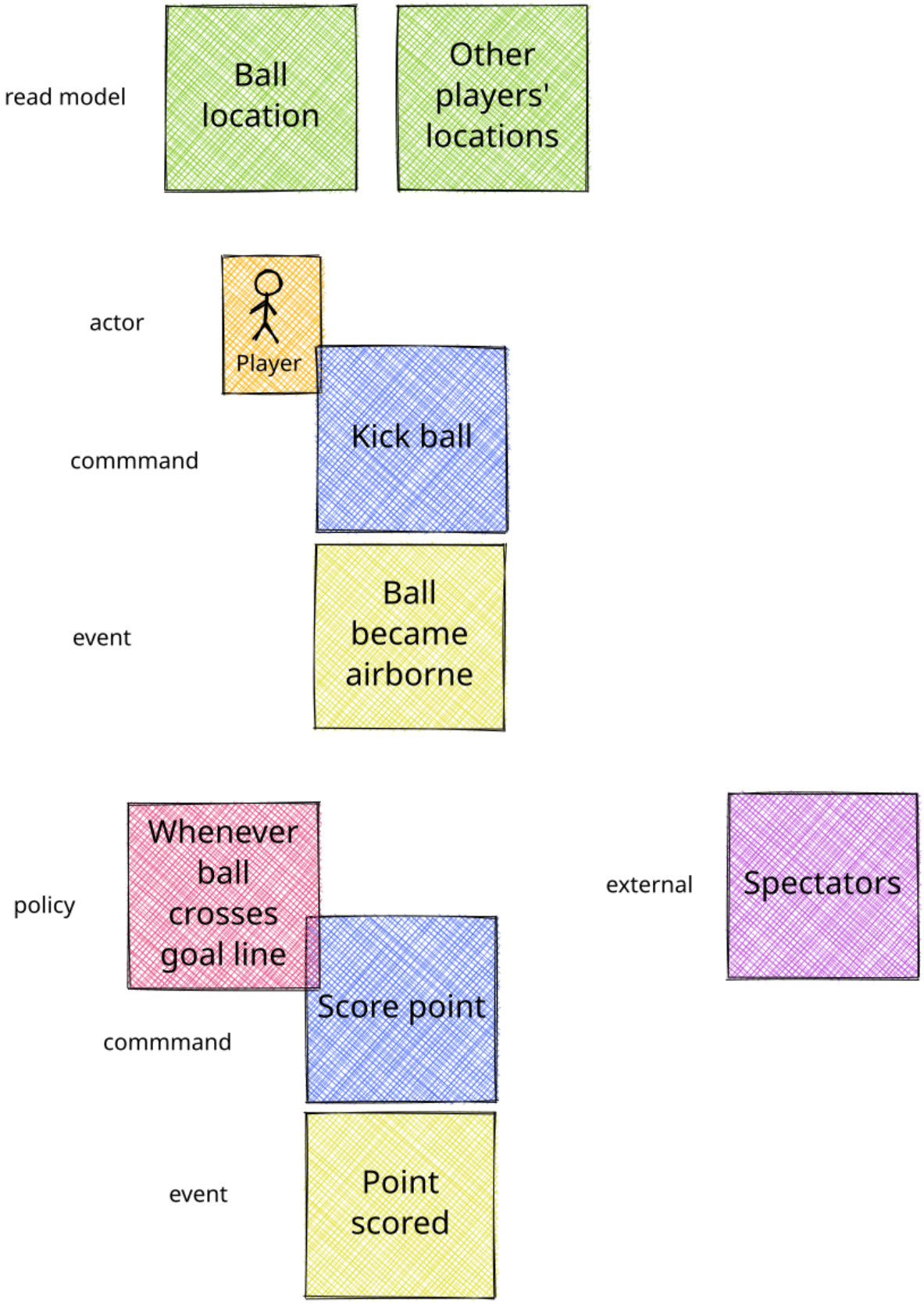
- We have a domain model, great!
- Added value comes from day-to-day conversations
  - Among developers
  - Between developers and the customer
  - Between developers and the user
- Is everyone speaking the same language?
- *Ubiqutous language*: All team members use important terms in the same way
  - Within a bounded context

6.1.4 Event storming workshop

- We need to quickly learn a new domain
  - *Business process modeling* and *requirements gathering*
  - Bring together *domain experts* and *developers*
  - Primary goal is a mutual understanding of the domain
- Discover events that occur in the business, and what triggers them
  - **Business Event**, e.g. *a customer has applied for a loan*
    - A change has occurred (in your business or in the real world)
  - **Command**, e.g. *create a new loan request*
    - A request or interaction to be made with a system (ours or external)
    - Submitted by a user, or by an external system
  - **Read model**, e.g. *customer account balance*
    - Information that a user or external system needs to base commands on
  - **Actor**, e.g. *loan requester*
    - Role of a person submitting a command
  - **Aggregate**, e.g. *Loan Application*
    - Entity(ies) of a business subdomain that should be viewed atomically
- Why do you think the focus is on *Events*, rather than *Aggregates*?

6.1.5 Event storming workshop (example)

- An small example of all concepts is here



6.2 Data design patterns

6.2.1 Idempotency

- Allow any failed or repeated action to be applied again
  - With the same result (if previously successful)
  - Without additional side effects that have business impact
- Example:
  - New user is stored in our database, but afterwards we failed sending their welcome mail (SMTP server was down).
    - Retry the database operation: User is already found, so instead we verify that the data matches
    - Retry sending the mail: We know that we didn't send the mail yet, so we send it once more
  - New user is stored, welcome mail is sent, but we failed updating our CRM system
    - Retry the database operation: User is already found, so instead we verify that the data matches
    - Retry sending the mail: We know that we've already sent this mail, so we simply do nothing
    - Retry updating the CRM system

6.2.2 Event Sourcing

- Traditional relational database: CRUD
  - Update in place
- Change log, shadow table
- Turn it upside down: *Event journal* is the source of truth
  - Read from the event journal to create your query model
  - No more CRUD
  - Read from your event journal again: *full-text search!*
  - Read from your event journal again: *business analytics!*
- Event journal can even be a part of your API

6.2.3 Eventual consistency

- Traditional approach to consistency (*transactions*)
  - Data store hides concurrent modifications of multiple areas from each other, enforcing constraints
  - Modifications typically (hopefully) fail if attempting to modify the same data
  - Even within one data store, hard to get 100% right
  - Complexity skyrockets when trying to scale beyond one data store (*distributed transactions, XA*)
- Eventual consistency
  - Embrace the flow of data through the system hitting data stores at different times
  - Embrace real time as a parameter to affect business logic
    - *Is it OK if a document I just saved doesn't show in the list until 0.5 seconds later?*
  - Apply **Idempotency** to all data store updates
  - Leverage **Event Sourcing** where possible

6.2.4 Command query responsibility segregation

- CQRS: Have two separate data models (and split your API accordingly)
  - A *command* model, for API calls that only change data (and do not return data)
  - A *query* model, for API calls that only return data (and do not change data)
- Builds on CQS (Command query separation). One method can only do one of two things:
  - Perform a *command*, by having side effects (and not returning a value)
  - Perform a *query*, returning a value (and not having side effects)
- We'll see CQS again

6.3 Data formats

6.3.1 XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Invoice
 xmlns="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2"
 xmlns:cac="urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2"
 xmlns:cbc="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2">
 <cbc:ID>42</cbc:ID>
 <cbc:IssueDate>2004-05-24</cbc:IssueDate>
 <cac:InvoiceLine>...</cac:InvoiceLine>
 <cac:InvoiceLine>...</cac:InvoiceLine>
 <cac:InvoiceLine>...</cac:InvoiceLine>
 <cbc:ShoeSize/>
 <cac:LegalMonetaryTotal>
  <cbc:PayableAmount currencyID="USD">52.00</cbc:PayableAmount>
 </cac:LegalMonetaryTotal>
</Invoice>
```

- Extensible Markup Language
- Composes very well
  - Namespaces prevent shadowing
  - Natural order of tags can be useful
- *De facto* schema standard (XSD) has unfortunate limitations
  - Hard to express "order does not matter"
  - Hard to express "this schema can be extended with extra tags and attributes"
  - Alternatives: *schematron* (alive) and *relax-ng* (dead?)
- Still, a very sensible default choice

6.3.2 JSON

```json
{
  "invoice": {
    "id": "42",
    "issueDate": "2004-05-24",
    "legalMonetaryTotal": {
      "payableAmount": {
        "value": "52.00"
        "currencyID": "USD"
      }
```

```
    }
  }
}
```

- *JavaScript Object Notation*
- Started its life in the web browser (~2000)
  - XML inconvenient to deal with in Javascript back then (SAX API)
  - JSON could just be parsed as Javascript directly
- No namespaces
  - JSON is useless without context
- No (useful) types
  - JavaScript *number* is a technically a double-precision float (even though in JSON it can contain unlimited digits)
  - Even JSON schema does not remedy this
- No comments

6.3.3 Protobuf

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

- Very compact binary format
- Started at Google, today >70 implementations
- Built with organic versioning in mind
- Ideal for storing events of event sourcing (if you have a lot of them)

6.3.4 Designing for extensibility

- Use schemes and code lists instead of fixed enumerations

```
<InvoiceAmount currencyID="USD">42.00</InvoiceAmount>
```

- Use rich data objects instead of flat numeric values
  - e.g. `Amount`, `Measurement`, `GeoCoordinate`, `Quantity`
- Use namespaces and URIs where you can

7 Break(out) 3

7.1 Let's hold an event storming workshop

7.1.1 Domain scope

- Let's find a domain and scope for the events we want to discover
  - Is there a shared system, or domain, most of you have worked on?
  - Is there a shared system, or domain, most of you know is important for your business?

7.1.2 Alternative: Pet Shop

(skip if domain is found)

- Let's model a pet shop!
  - Our family owns a pet shop, which has a building that houses pets for sale
  - We regularly sell pets, and re-stock
  - Pets need to be fed
  - Some pets have special other needs
  - We only want to house cute pets!
- We want to automate as much as we can, and hence hold an event storming workshop

7.2 Event storming stages

7.2.1 Events

- Distribute orange post-its
- Remember, an event is in PAST TENSE, e.g.: *Missiles have been launched User has subscribed to newsletter*
- Designate a wall as space
  - Time flows roughly from left to right (where relevant)
- Start with "pivot" event in center
- Write other events that come to mind
  - Order with existing events, keeping time
  - Feel free to rename as discussions occur

7.2.2 Exploring our events

- Distribute blue, yellow, green and pink post-its
  - Blue: *command*
  - Yellow: *actor*
  - Pink: *external system*
  - Green: *read model*
- Remember, a command is in IMPERATIVE, e.g. *Launch missiles Register user subscription request*
- Select important events, that related to something a system could do for us
  - What command could cause this event? (blue)
  - Who or what can trigger this command?
    - Who: Actor (yellow)
    - What: External system (pink)
  - What information is needed to construct the command (green)

7.2.3 Aggregates

- What nouns have we discovered that are good candidates for aggregates?
  - Group the commands by aggregate
- What aggregates would be good candidates for microservices?

8 Wrapping up today

8.1 Let's do another round

8.1.1 Please share!

- Name one thing that you learned
- Name one thing that you knew already
- Name one thing that surprised you

9 Start of day 2

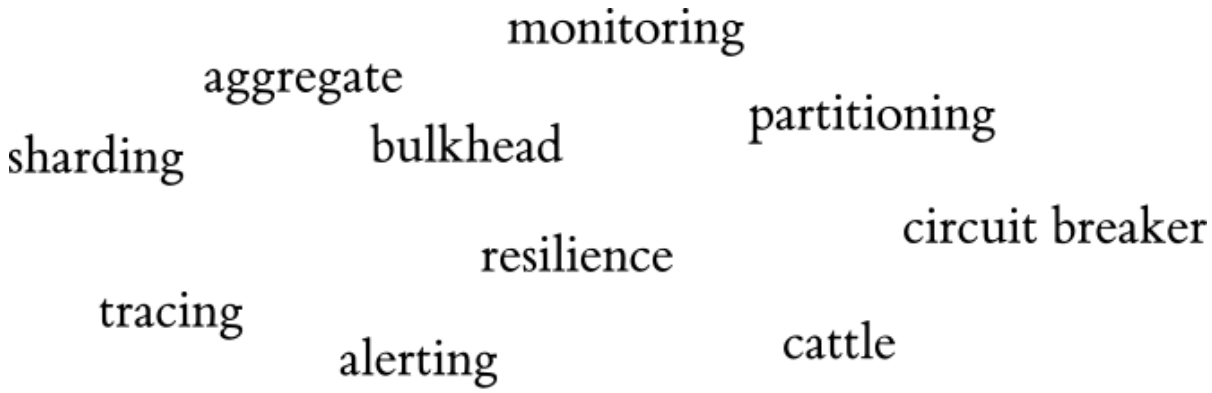
9.1 Getting started

9.1.1 Schedule, Day 2

Time	Duration	Activity	Weight
09:00	00:10	Welcome, Outline/Agenda	
09:10	00:10	Recap of day 1	
09:20	00:20	Getting your service used	
09:40	00:40	An introduction to REST	
10:20	00:30	(Break)out 4	
10:50	00:20	Discussion of breakout results	
11:10	00:60	REST patterns	
12:10	00:30	Lunch	
12:40	00:10	REST API examples	
12:50	00:30	(Break)out 5	
13:20	00:15	Discussions of breakout results	
13:35	00:20	Microservice life cycle	
13:55	00:10	Security architecture	
14:05	00:15	Strategy and team dynamics	
14:20	00:30	Group exercise (adoption)	
14:50	00:30	(Break)out 6	
15:20	00:30	Presentations of breakout results	
15:50	00:10	Wrap-up, reserved time for extra subjects	

9.1.2 Recap of day 1

Who can tell us something about:



10 Getting your service used

10.1 Public API

10.1.1 An API is an interface

- *Application Programming Interface*
  - It's how external components affect what our service does
  - Better lay down some rules
- But our service is only used by our team, we don't need documentation!
- Ideal for test-first development
- Where do I put my private API?

10.1.2 Example API

- Let's look at an example API example API together
  - Its RAML source is available
- Semantic format for describing REST APIs: RAML, OpenAPI
  - RAML: YAML-based, better re-use, easier to write by hand
  - OpenAPI: JSON/YAML-based, more popular

10.1.3 Content-type negotiation

- Embrace content-type negotiation (XML *and* JSON, not XML *or* JSON)
- XML API:
  - Do create XSD for your data types, but communicate how it should be interpreted
  - Do you reserve the right to add new tags and attributes?
- JSON API:
  - Create JSON schemas for everything
  - In addition, verbosely describe all numeric types and their intended usage

10.2 Public developer guide

10.2.1 But I've written the documentation!

- Just a list of endpoints may not be enough for some developers
- Lot of context and assumed knowledge
  - Ubiquitous language may not extend to all new API users
  - Lack of experience with JSON, XML, HTTP headers

10.2.2 Different people, different learning styles

- Write a developer guide that describes typical scenarios from a user's perspective
  - How to get started (e.g. get an SSL certificate)
  - How to list widgets in XML or JSON
  - How to create a new widget
- There's no shame in taking an English technical writing course
- Pick tooling that suits your way of working (e.g. `HTTPie`, `org-mode` with `org-babel`, ...)

10.3 Public service dashboard

10.3.1 Priorities!

- What's the first thing you do when you get to your office?
- Users will be curious about your service status
  - If your users are internal, give them access to the actual dashboard
  - In fact, consider giving them access to your source code and issue tracker as well

10.3.2 Designing your dashboard

- Your dashboard should be showing
  - System metrics (load average, disk space, CPU usage, memory usage, network I/O, disk I/O)
  - Your process' metrics (CPU usage, memory usage)
  - Your VM's metrics (Heap committed, heap used, GC time, thread count, log count)
  - Your framework's metrics (HTTP server open connections, HTTP client open connections, response times, response errors)
  - Your business metrics (number of pets signed up, total invoice amount, size of received chat messages)
- For each environment, after a few days examine the graphs
  - Establish a baseline, and create an alert for *each* metric

11 An introduction to REST

11.1 REST philosophy

- **World-wide web** (1990): HTTP over internet, with hypermedia (HTTP)
  - Unprecedented scaling
  - Applications (e.g. Facebook, Amazon) can develop continuously without clients (browsers) breaking
    - *(at least, until they figured out native clients means no ad-blockers...)*
  - Managed to survive 20+ years in a wild changing landscape, with limited technical debt
    - Most of HTTP and HTML are still relevant
- Apparently, it's possible to perform heterogeneous systems integration without any
  - legal contracts,
  - deep specifications, or
  - personal knowledge
- Try pulling that off in your enterprise!

11.2 REST principles

- Apply the WWW success for system-to-system communication
  - `RE` presentational `S` tate `T` ransfer
- Request-based from *client* to *server*
  - Distinctly separated roles that two systems or actors play when handling a request
- Stateless
  - Request contains all information needed to process it (instead of, e.g. the TCP connection socket)

- Caching
  - Responses must clearly state, and have sensible defaults, on how content can be cached
- Uniform interface
  - All components are accessed the same way
- Layered system
  - Intermediaries can be transparently inserted between client and server (load balancers, proxies, security gateways, ...)

11.3 Resources

In REST, the *client* accesses a *resource* on the *server*, through a *request*.

A resource:

- Is a noun, e.g. *user*, *invoice*, *setting*, but also *transaction*, *order status*, or *deletion process*
- Can have several representations, e.g. XML, JSON, HTML, picture, small, large
- Is accessed through one or several URLs
  - `/users/15`, `/users/latest`, `/users?name=Santa` might all return the same resource
- Is interacted with through a limited set of verbs (more on that later)

Remember your event storming workshop?

11.4 An introduction to HTTP

- Text-based protocol over TCP
  - Client sends a request (with *verb*, *headers*, and optional *body*)
  - Server sends a response (with *status line*, *headers*, and optional *body*)
  - (since HTTP 1.1) Client sends a new request, etc.

Client sends:

```
GET /cats/latest?fur=white&size=small
Accept: image/png
User-Agent: Mozilla/5.0
```

Response then comes in:

```
200 OK
Content-Type: image/png
Content-Length: 53748

[...kitten goes here...]
```

11.4.1 HTTP Verbs

- Predefined verbs imply important caching and retry semantics

Verb	Safe to retry?	Idempotent?	Request body?	Response body?	Cache response?
GET	yes	yes	-	yes	yes
HEAD	yes	yes	-	-	yes
PUT	-	yes	yes	-	-
POST	-	-	sometimes	sometimes	-
DELETE	-	yes	-	-	-

- Why wouldn't `PUT` or `DELETE` be safe to retry?
- Rest is not RPC

11.4.2 Example HTTP status codes

The *status line* contains a code and then a short description. The description is not predescribed, and sometimes contains useful information.

- `200 OK`
  - The request succeeded. Typically a response body is present.
- `201 Created`
  - The request succeeded, and a new resource was created as a result.
- `204 No Content`
  - The request succeeded, but no content is available.
- `302 Found`
  - The resource was found at a different URL, which is returned in the `Location` header.
- `404 Not Found`
  - The resource does not exist. This does *not* necessarily mean that an API endpoint does not exist.

This is not a full list. See the HTTP official status codes or a more graphically-accessible variant.

11.4.3 Example HTTP Headers

- `Accept: image/*`
  - Sent in a *request* to indicate the MIME types that the client prefers for this request (but there's no guarantee)
- `Content-type: image/png`
  - Sent in a *request* or *response* to indicate the actual MIME type of the body
- `Content-length: 5124`
  - Sent in a *request* or *response* to indicate the size of the body in bytes (if known)
- `Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT`
  - Sent in a *response* to indicate when that resource was last changed

This is not a full list.

11.5 REST API design

- Find resources for your domain
  - Perhaps using an *Event Storming* workshop (from *Domain-Driven Design*)
- Use CQRS (Command Query Response Segregation)
  - Find representations for those resources (current state and/or events): `GET`
  - Find commands affecting those resources (creation, modification, transactions): `POST`, `PUT`, `PATCH`, `DELETE`
- Size limits on everything (do we need to stream or read it in memory?)
- XML, JSON, CSV, text, protobuf (more content-type negotiation later)
- Decide on a *Service Level Objective* for your API (yes, already now)

12 Break(out) 4

12.1 Finding REST services

12.1.1 Assignment

- Divide into teams
- Find documentation of a REST web service
  - Preferably: Internally published in your company
  - Otherwise: Publicly on the internet, examples: *e-conomic*, *twitter*, *github*, *AWS*, *flickr*
- Create a full example request
  - Request headers and (if relevant) body
  - Response status line, headers, and (if relevant) body
  - Bonus points if you can actually execute the request!

12.1.2 Presentation of results

- Show us the services you found
- How did you find the quality of documentation?

13 A selection of REST patterns

13.1 Resource tags and caching

13.1.1 Resources have versions

- Servers can include an `ETag`, which specifies which *version* of a resource is being served

```
GET http://example.com/widgets/15

200 OK
Content-Type: application/json
ETag: "524"
```

- No guarantees are made about the content of `ETag`, but often APIs will document what it represents, e.g.
  - A timestamp of some sort
  - A monotonically-increasing number
  - A hash of the latest content

13.1.2 Conditionally retrieving a resource

- If the latest `ETag` we have seen is "524", we can poll for changes
- The `If-None-Match` header will *only* execute our request if the `ETag` has changed

```
GET http://example.com/widgets/15
If-None-Match: 524

304 Not Modified
```

- The server will not send any response if the resource is still at this version

13.1.3 Optimistic offline lock

- The `ETag` is also useful to make sure nobody else has edited a resource that we're writing back
- The `If-Match` header will *only* execute our request if the `ETag` matches

```
PUT http://example.com/widgets/15
If-Match: 12345
Content-Type: application/json

{ /* ... some content ..* }

412 Precondition Failed
```

13.1.4 Resources can be modified

- Servers can include a `Last-Modified` tag, which specifies *when* a resource was last changed
- This can be useful in addition to an `ETag` tag

```
GET http://example.com/widgets/15

200 OK
Content-Type: application/json
ETag: "524"
Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
```

- Request header exist that perform checks against the last-modified date, like `ETag`:
  - `If-Modified-Since` executes the request *only* if the last-modified is past the given date
  - `If-Unmodified-Since` executes the request *only* if the last-modified is at most the given date

13.1.5 Preventing caching

- For service-to-service REST calls, we generally don't worry about caching
- For web browsers, we often want to disallow caching of REST responses
  - Include `Cache-Control: no-cache`

13.2 Content-type negotiation

13.2.1 Resource representation

- The same REST URI is allowed to have several representations
  - XML, JSON or Protobuf
  - Short or long
  - Version 1 or version 2

13.2.2 Specifying resource representation

- The server specifies the representation of a resource
  - The `Content-Type` resource header
- This is typically a well-known value
  - `text/xml`
  - `application/json`
  - `application/protobuf`

- But it doesn't have to be
  - `application/vnd.example.myresource.v1+json`
  - `application/vnd.example.myresource.v2+json`
  - `application/vnd.example.myresource.short+json`
  - `application/vnd.example.myresource.long+json`

13.2.3 Requesting a resource type

- The client sends an `Accept` header with the representations it wants/understands
- In case of a single representation:

```
GET http://localhost/myresource
Accept: application/json
```

- In case multiple representations are alright (order has no semantic meaning):

```
GET http://localhost/myresource
Accept: application/json, text/xml
```

- Multiple representations are alright, but preference for xml:

```
GET http://localhost/myresource
Accept: application/json;q=0.9, text/xml
```

13.2.4 Serving resource alternatives

- Content-type negotiation is complex to implement
- How easy it is to support depends on your framework
  - *Spring Boot* has many different ways to manage resource representation
    - Look into `HttpMessageConverter`, so you can take control
  - Others, e.g. `akka-http` has a marshaling infrastructure that directly models content-type negotiation

13.3 Asynchronous and long-running processes

13.3.1 Case: REST API to represent workflow instances

- Start a new workflow
- See which human is working on the case
- Quickly resume if system is working on the case

13.3.2 REST is about resources

- For slow-running processes, make the process itself a resource, e.g.
  - `/workflows/`
  - `/transactions/`
  - `/cases/`
- You can now reason about individual processes
  - Query state, affect them, delete them, see changes

13.3.3 Observing change on one resource

- Tell client to periodically poll
  - Use `If-None-Match` for early exit
  - Use heavy caching on the server-side to reply to polls as early as possible

13.3.4 Observing change on a set of resources

- Build your system using *Event Sourcing*
- Expose your event journal (or a light, or filtered version) as a REST resource
  - This can be done regardless of storage (JDBC, Cassandra, Kafka)
- Various candidates for the data format
  - Plain

```
GET http://localhost/journal/events?since=Wed+May+26+11:59:05+2021+UTC&limit=50
Accept: application/json
```

- Hanging GET

```
GET http://localhost/journal/events?since=Wed+May+26+11:59:05+2021+UTC&limit=50&maxwait=60000
Accept: application/json
```

- Server-sent events (SSE)

```
GET http://localhost/journal/events?since=Wed+May+26+11:59:05+2021+UTC
Accept: text/event-stream
```

- Web sockets

```
GET http://localhost/journal/events?since=Wed+May+26+11:59:05+2021+UTC
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

13.4 Multi-dimensional versioning

13.4.1 Semantic versioning in REST

- Often used for library dependencies and packaged software releases
- Version number has three parts (*major*, *minor*, *patch*): version `2.0.15`
  - A new release always must have a new version
  - If a release has no new functionality (only bugfixes), increase the *patch*
  - If a release has new functionality that doesn't break API promises, increase the *minor*
  - If a release has new breaking functionality, increase the *major*
- How does this relate to REST?

13.4.2 Semantic versioning in REST (cont.)

- How does this relate to REST?
  - It doesn't!
  - REST is a call to a remote system
    - Could be deploying new versions multiple times per day
  - The whole point is the client *doesn't* want (or need) to see those
- OK, what do we do instead?
  - Version across all HTTP dimensions

13.4.3 Versioning in body structure

- Many extensions fit fine into existing body structure
  - Adding of fields
  - Adding of values to enumerations or code lists
- If DDD has done its work, terminology should mostly hold

13.4.4 Versioning in content type

- If a breaking change is needed
- It might be limited to only one content type
- Client requests old version:

```
GET http://localhost/myresource
Accept: application/vnd.example.myresource.v1+json
```

- Client requests new version:

```
GET http://localhost/myresource
Accept: application/vnd.example.myresource.v2+json
```

13.4.5 Versioning in query parameters

- Don't do this
  - Query parameters affect *which* and *what* resource(s) are returned, not *how*
- The meaning of query parameters may themselves be versioned

13.4.6 Versioning in path

```
GET http://localhost/service/versions/1/myresource
```

- Often used as first choice
- Should be your last resort:
  - Your path is the name of your resource
  - Your DDD workshop (probably) didn't event storm about "versions"
  - Your system (probably) doesn't have 2 complete implementations
  - This does often not reflect reality

13.4.7 Versioning using custom headers

- Client sends a custom header of the API version they've implemented against
- Server sends a custom header of the API version that's current
- This does kinda work
- Fairly weak way to work around *actually* dealing with semantic changes and compatibility

14 REST API Examples

14.1 Examples of REST design

14.1.1 Github

```
GET https://api.github.com/search/issues?q=windows+label:bug+language:python+state:open&sort=c
↪  reated&order=asc
Accept: application/vnd.github.text-match+json

200 OK
Content-Type: application/vnd.github.text-match+json
{
  "text_matches": [
    {
      "object_url": "https://api.github.com/repositories/215335/issues/132",
      "object_type": "Issue",
      "property": "body",
      "fragment": "comprehensive windows [...] ter.\n",
      "matches": [ ... ]
    }, [...]
  ]
}
```

14.1.2 Github: Search for issues

Notes:

- Using a custom content-type to indicate a special flavor of JSON
- Relying on `GET` to indicate a read request

14.1.3 AWS

```
GET https://ec2.amazonaws.com/?Action=RunInstances&ImageId=ami-2bb65342&MaxCount=3&MinCount=1&
↪  Placement.AvailabilityZone=us-east-1a&Monitoring.Enabled=true&Version=2016-11-15&X-Amz-Alg
↪  orithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIDEXAMPLE%2F20130813%2Fus-east-1%2Fec2%2Faws4_r
↪  equest&X-Amz-Date=20130813T150206Z&X-Amz-SignedHeaders=content-type%3Bhost%3Bx-amz-date&X-
↪  Amz-Signature=525d1a96c69b5549dd78dbbec8efe264102288b83ba87b7d58d4b76b71f59fd2

200 OK
[... lots of json ...]
```

14.1.4 AWS: Create EC2 instance

Notes:

- a `GET` verb is used to have side effects!
- No resource representation of the actual server to be created
- Proprietary authentication mechanism, and using the URL for this

14.1.5 Flickr

```
GET http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=xxx&text=trysil&f
↪  ormat=rest&auth_token=xxx&api_sig=xxx
Accept: text/xml

200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
    <photos page="1" pages="121" perpage="100" total="12050">
        <photo id="12185296515" owner="110367434@N02" secret="7bf83bc507" server="3714"
        ↪  farm="4" title="#wall #clock #wood #old #Norway #Trysil #travel #travelling"
        ↪  ispublic="1" isfriend="0" isfamily="0" />
        <photo id="12185880206" owner="110367434@N02" secret="c8042c1764" server="7382"
        ↪  farm="8" title="Good morning!   #Norway #Trysil #window #snow #beautiful
        ↪  #landscape #travel #travelling #polar #expedition" ispublic="1" isfriend="0"
        ↪  isfamily="0" />
        ...
        <photo id="11793639173" owner="40644602@N08" secret="ba2bdabf5c" server="7451"
        ↪  farm="8" title="by beateorten http://ift.tt/1dHDdQL" ispublic="1" isfriend="0"
        ↪  isfamily="0" />
    </photos>
</rsp>
```

14.1.6 Flickr's image search

Notes:

- Overloading of methods in the request URL
- Overloading of content type in the request URL
- Overloading of HTTP status code in the response
- No obvious way to explore the API further (how do I load a photo?)

15 Break(out) 5

15.1 Designing an API

15.1.1 Write a RAML or OpenAPI description for a pet store API

- We're writing a pet store automation system, and need to be able to register, find, and sell pets.
- We need APIs to:
  - Register a newly purchased pet
  - Register the feeding of pets
  - Search pets according to customer preferences
  - Register the sale of a pet
- RAML
  - example: https://raml.org/developers/raml-200-tutorial
  - spec: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md
  - online editor: https://raml-org.github.io/playground/learn_raml.html

- OpenAPI
  - spec: https://spec.openapis.org/oas/latest.htmla
  - online editor: https://editor.swagger.io/

15.1.2 Discussion

- One team makes their RAML or OpenAPI file available
- *Another* team then tell us how to:
  - Register a newly purchased pet
  - Register the feeding of pets
  - Search pets according to customer preferences
  - Register the sale of a pet

16 Micro service life cycle

16.1 Dependency management

16.1.1 Developing a new service

- I want to write a new micro service!
  - I need a database, a queue, the filesystem for some caching
  - Oh, and I'm talking to twitters API, and our home-grown analytics API
- How do I deal with these dependencies during day-to-day development?
  - "Leaf" dependencies: often OK to run directly (e.g. data stores)
  - "Node" dependencies (other microservices): often have dependencies of their own
    - You know its API, right?
    - Mock it! Wiremock, or any simple http server

16.1.2 Running dependencies

- Maintain a `docker-compose` file for your project
  - Real dependencies: they're probably on `docker-hub` already
  - Mocks: use the `build` feature if needed
- New developers can get started instantly

16.2 Extending a service

16.2.1 Developing a new feature

- Don't hide your new feature on a branch
- Release early and often
  - But only activate it in certain environments and/or users
- Feature flag
- A/B testing

16.3 Testing

16.3.1 Unit tests OK, Integration tests not

graphics/tests.mp4

16.3.2 Introducing bugs

- Rate of bugs introduced into systems are a function of
  - Developer experience
  - Development environment (physical and technological)
  - Methodology

16.3.3 Finding bugs

- Fixing bugs is more expensive, the later they are found
  - While writing code: just think of different solution
  - While code is in review: communication, context switch, and the above
  - While code is in user testing: (much) more communication, context switch, and all the above
  - After code is released: (even) more communication, impact analysis in data, and all the above

16.3.4 Preventing bugs

- Test at different layers
  - On code itself: Pair programming
  - On one unit (e.g. class): *Unit tests*. Run in seconds.
  - On one service (e.g. rest API): *Component tests*. Run in tens of seconds.
  - On a suite of services (e.g. UI): *End-to-end tests*. Run in minutes.
  - On your entire infrastructure: *Smoke tests*. Run periodically, on production, with external dependencies

16.4 Deployment

16.4.1 Getting your service out there

"*All software has a test environment. Some software is lucky to have a separate production environment as well.*"

- unknown

16.4.2 Doing deployments

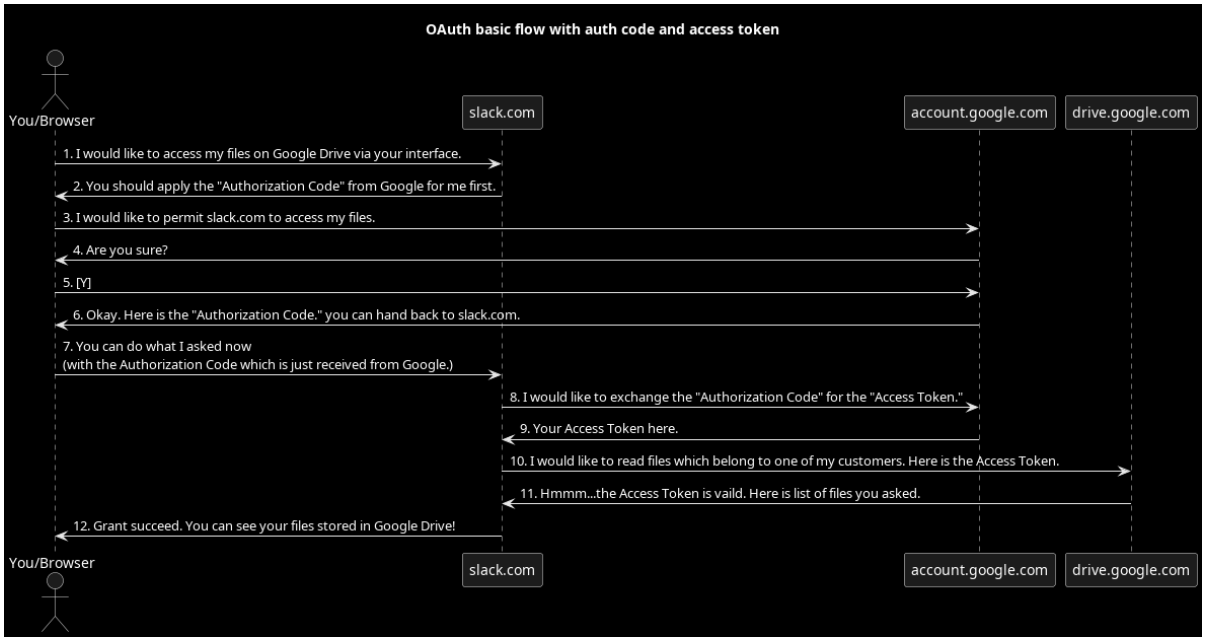
- Automate the environments themselves (`terraform`, `vagrant`, ...)
- All deployments to all environments must be automated
- It's OK to have gatekeepers, e.g.
  - After a PR is merged, automatic deploys are done to `dev` and `test` environments
  - The `prod` environment requires a manual button press
- Forward deploy only
  - Rollbacks are a pain
  - Your next deploy is only minutes away
  - Emergencies should be rare (testing, early release, multiple environments)

17 Security architecture

17.1 Authentication patterns

17.1.1 User-to-service authentication

- I want code running on a user's computer to call me (let's assume web browser)
- OpenID Connect, simplified flow:
  1. *Resource owner* wants *client* to log on to *authorization server*
  2. Client is redirected to authorization server
  3. User verifies trust of authorization server and logs on
  4. Authorization server redirects client back (with authorization code)
  5. Client contacts resource owner with *code*
  6. Resource owner exchanges code for *token*
  7. Token can be used in `Authorization: Bearer` http header



17.1.2 Service-to-service authentication

- I want code running on other backend services to call me (outside of the context of a user)
- Mutual TLS
  - Server has a certificate, proving it's who it claims
    - Client has established trust on a root certificate, having signed the server certificate
  - Client has a certificate, proving it's who it claims
    - Server has established trust on a root certificate, having signed the client certificate
- In practice
  - Create (or purchase) a root certificate for your business, lock it tight
  - Create intermediate CAs for particular roles, e.g. for singing micro-services
  - Use *Certificate Signing Requests* to reflect real business flow
  - For your service clients
    - Have server sign client certificates directly
    - Or, delegate to an intermediate CA, and implement whitelisting
- Confirm that OCSP (*Online Certificate Status Protocol*) can be used to revoke certificates

17.2 Implementation

17.2.1 Authorization checks

- Prefer to keep internal to service
- Replicate user memberships through event sourcing
- Synchronous calls least favorable choice

18 Strategy and team dynamics

18.1 Succeeding with microservices

18.1.1 Microservices and agile

- Embrace change
- Team visibility
- Stakeholder support
- Team(s) in same time zone as stakeholders (which includes users)
  - Distributed users? distributed team!
- Conway's Law

18.1.2 Migrating your monolith

- Chainsaw anti-pattern
- Strangler pattern
- Modules

18.1.3 Do we need a separate dev/ops team? (no)

- Automate everything (rolling production deploy)
- Deploy in the morning, monitor your dashboards
- However, "infra tooling" or "platform" team can be helpful
- The same holds for the "DBA" team

18.2 Group exercise

18.2.1 Microservice adoption brainstorm

- Distribute post-its
- Write one post-it for: *In my daily work, I expect THIS to be most helpful in writing microservices*
- Write one post-it for: *In my daily work, I expect THIS to be the biggest blocker for writing microservices*

18.2.2 Gather results

- Two white board sections
  - *Drivers*
  - *Challenges*
- Put up your post-it, read aloud, and explain

18.2.3 Discussion

- Are there patterns to the drivers and challenges?
- What can we do to retain and strengthen the drivers?
- What can we do to remove the challenges?

19 Break(out) 6

19.1 Finding microservice candidates

19.1.1 Brainstorm

- Are there monoliths or other systems related to you or your team, that could benefit from microservices?
- Lets create a list of *large* systems that you know of:

System name	Developer count	Lines of code

- Hint: Lines of code `git ls-files | xargs wc -l`
- Hint: Developer count `git shortlog -s -n --all`

19.1.2 Strategy

- Which of these systems have changes planned in the next year?
- Which of these systems have frequent bugs?
- Which of these systems do developers hesitate to make big changes to?

System name	Activities

19.1.3 Design

- Divide into teams
- Pick one system and activity, and design a microservice that implements part of that domain
  - What surrounding data stores do you need to create?
  - How do the existing system and the microservice talk to each other?
  - Who maintains the existing system and microservice going forward?

20 Interesting links

https://world.hey.com/joaoqalves/disasters-i-ve-seen-in-a-microservices-world-a9137a51 https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1

21 Notes

- Add rabbitMQ stream example
- Pure function example (split up business logic and side effects)
- Screen sharing of others??
- Draw the UML diagram from https://developer.okta.com/blog/2019/08/22/okta-authjs-pkce
- Add HTTP cats https://http.cat/401
- Add comics