# Microservices and Domain-Driven Design

Jan Ypma

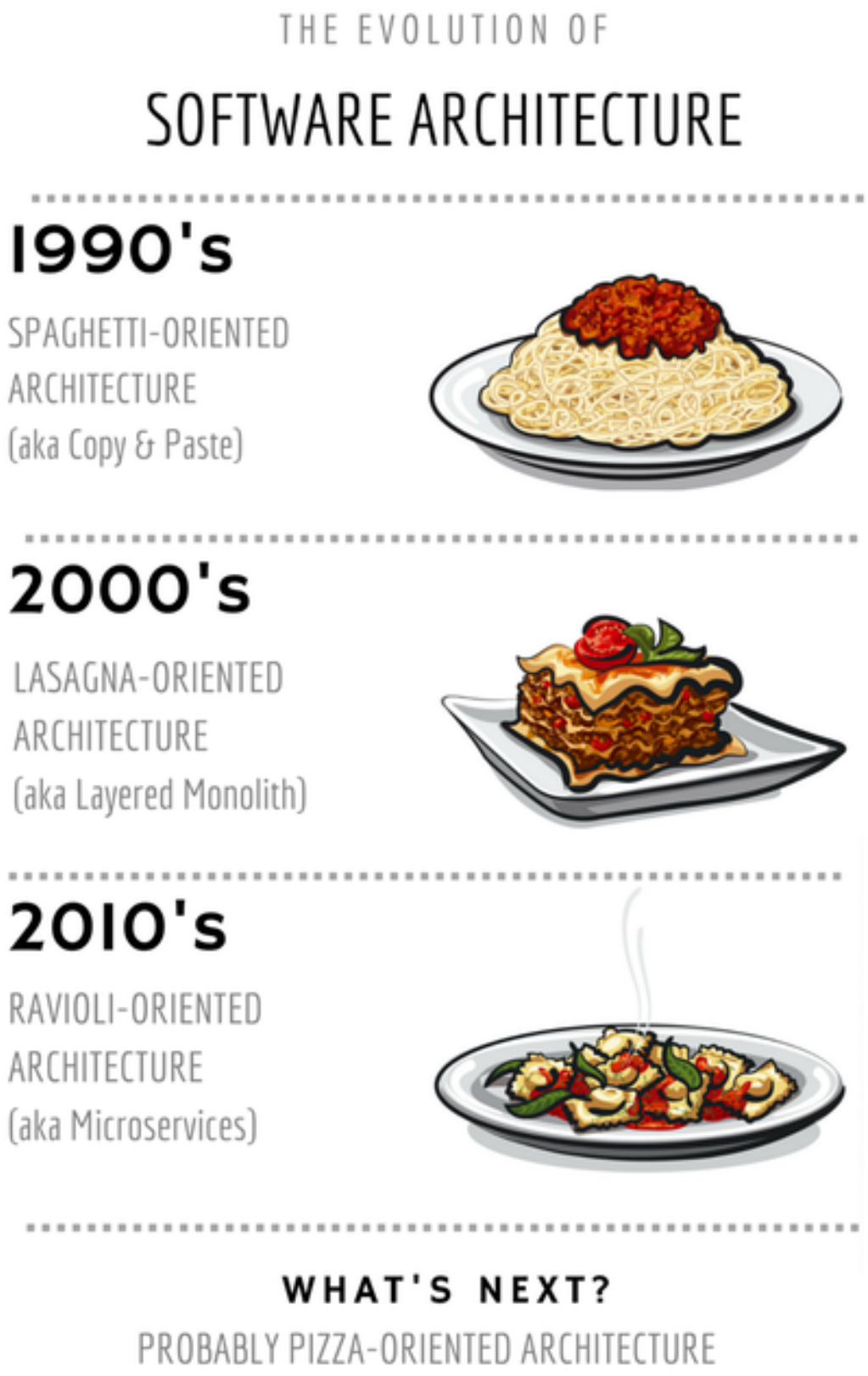
September 6, 2023

## Contents

# 1 Introduction

## 1.1 Getting started

### 1.1.1 Welcome



### 1.1.2 About me

- Jan Ypma
  - Independent software architect
  - Java, Scala, Groovy, C++, Rust, Lisp
  - Contributor to various open source projects
  - Fan of functional programming and distributed systems
  - Agile coach and fan of Domain-Driven Design
  - `jan@ypmania.net`, https://linkedin.com/in/jypma

### 1.1.3 Agenda

| Time | Duration | Activity | Weight |
|---|---|---|---|
| 09:00 | 00:05 | Introduction | |
| 09:05 | 00:10 | Resilience | |
| 09:15 | 00:15 | Infrastructure architecture | |
| 09:30 | 00:15 | Data architecture | |
| 09:45 | 00:05 | Domain-Driven Design | |
| 09:50 | 00:15 | Event Storming | |
| 10:05 | 00:20 | REST API methodologies | |
| 10:25 | 00:05 | Wrap-up, reserved time for extra subjects | |

## 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

$$P(everything\ working) = P(one\ service\ is\ working)^n \tag{1}$$

*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 Measure service quality

- Service Level Indicator (SLI)
  - An actual number that indicates something that relates to service quality
    - 99th percentile response time
    - 24-hour window success rate of HTTP requests
- 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 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

### 2.2.2 Recommendations

- 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

## 2.3 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

## 2.4 Guidelines

### 2.4.1 Microservice pitfalls

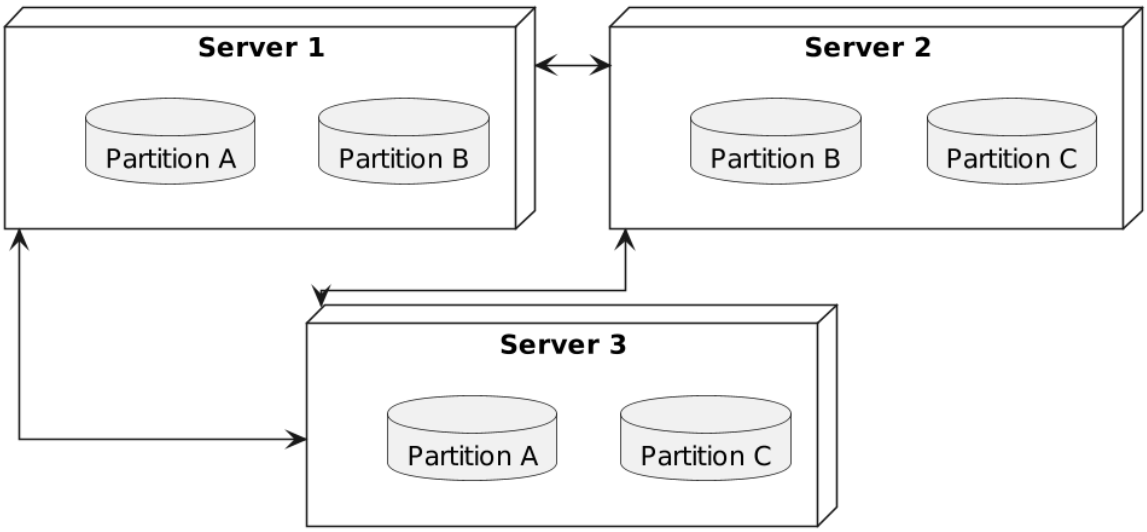
- Service co-dependencies
  - Keep HTTP calls one way only
  - Plugin pattern
- Nested synchronous service calls
  - Added latency and failure possibility
  - Avoid these with event sourcing
  - Replicate data instead, or call asynchronously when possible

# 3 Infrastructure architecture

## 3.1 Partitioned data stores

### 3.1.1 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




### 3.1.2 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 (with Cassandra API), Keyspaces
- Google: BigTable
- Azure: Cosmos DB (with Cassandra API)

### 3.1.3 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 ( * ), Event Hub

( * ) *not partitioned, size-limited*

### 3.1.4 Partitioned search

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

- Open source: Opensearch, SoLR
- Amazon: Hosted opensearch
- Google: Hosted elasticsearch
- Azure: Hosted elasticsearch

## 3.2 Monitoring and alerting

### 3.2.1 Introduction

- Logging need not be a cross-cutting concern
  - Create monitored metrics instead
  - Pro tip: monitor your log level counters
- 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

### 3.2.2 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

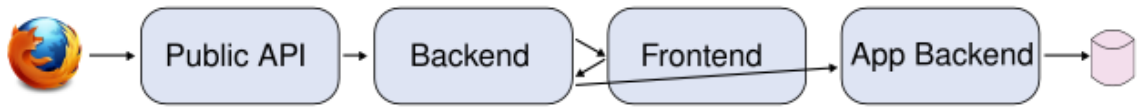
#### 3.2.3 Example dashboard

- You can use play.grafana.org with some example data

## 3.3 Request tracing

### 3.3.1 Complex service dependencies

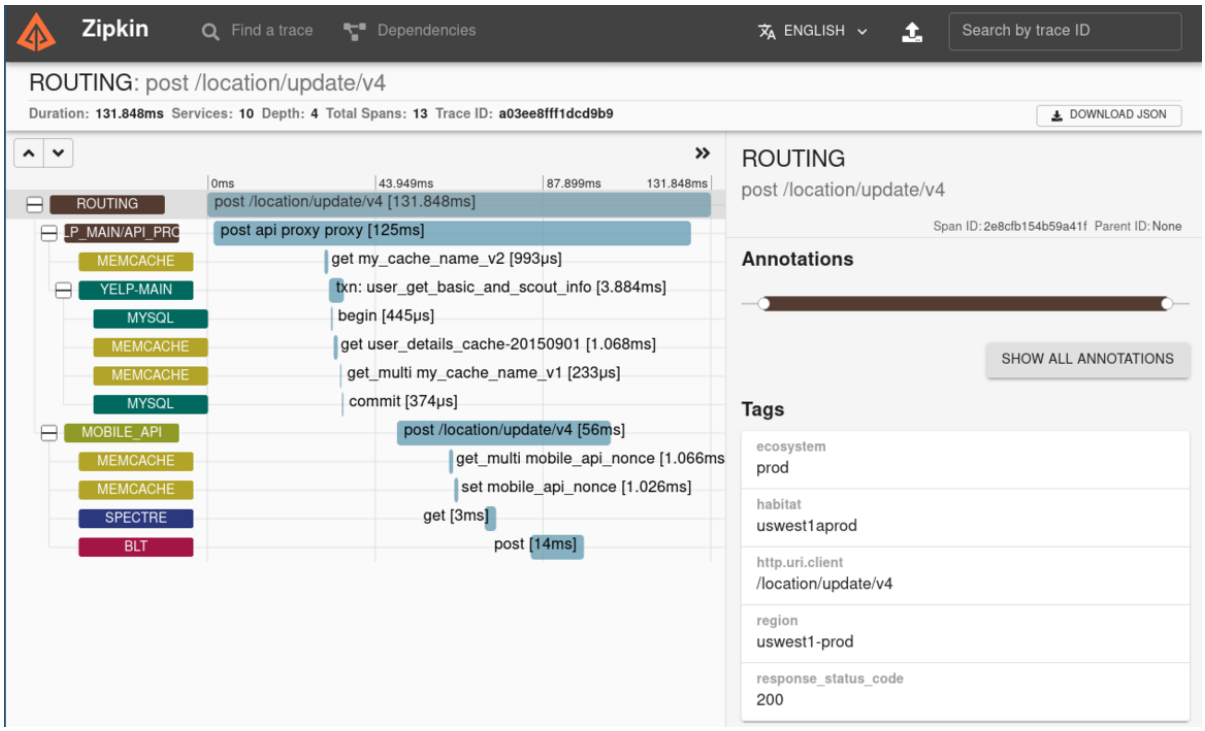
- Services can have complex calling stacks




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

### 3.3.2 Two mature solutions

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




## 3.4 Deployment

### 3.4.1 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

### 3.4.2 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"]
```

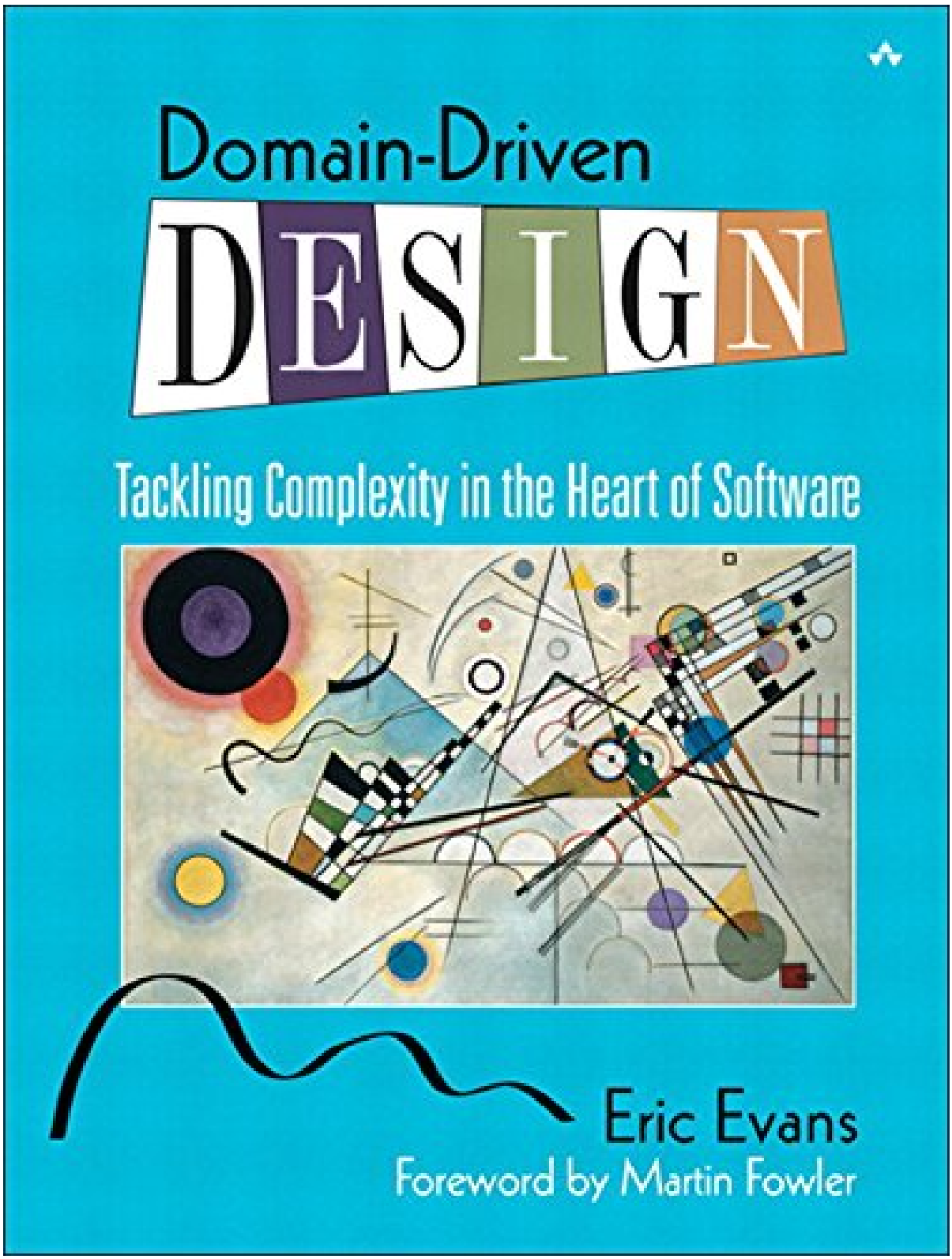
- Layers
- Volumes
  - Handling of persistent data
- Port mapping
- User mapping
- Don't run as root

### 3.4.3 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
- Configure your runtime to create a heap dump on out of memory (this *will* happen)

# 4 Data architecture

## 4.1 Domain-driven design



### 4.1.1 Introduction

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

### 4.1.2 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

### 4.1.3 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

### 4.1.4 Implementation patterns

DDD defines several patterns that are relevant as software architecture during implementation.

- *Entity*: A mutable representation of one business concept
- *Value Object*: An immutable set of data going to or from an *Entity*
- *Aggregate*: A group of *Entities* that should always have a consistent view

Have you implemented similar patterns? What did you call them?

### 4.1.5 Event storming workshop

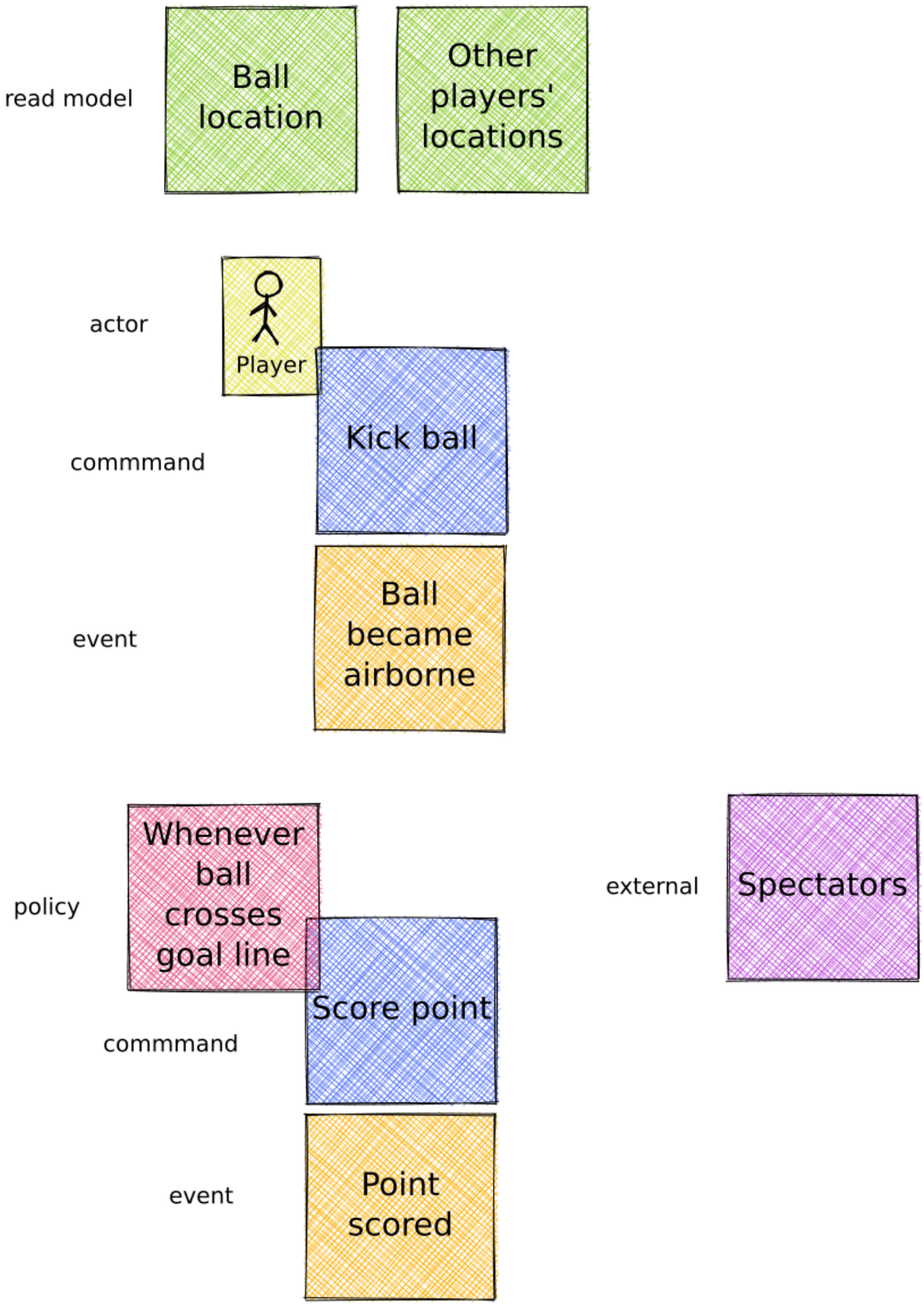
- We need to quickly learn a new domain
  - *Business process modeling* and *requirements gathering*
  - Bring together *domain experts* and *developers*: Invite the right people!
  - Primary goal is a mutual understanding of the domain
- Alberto Brandolini (2012): Event Storming

#### 4.1.6 Event storming elements

- 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)
    - Decided upon and initiated 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*?

#### 4.1.7 Event storming workshop (example)

- An small example of all concepts is here


read model
Ball location
Other players' locations
actor
Player
commmand
Kick ball
event
Ball became airborne
policy
Whenever ball crosses goal line
commmand
Score point
event
Point scored
external
Spectators

## 4.2 Data design patterns

### 4.2.1 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

### 4.2.2 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

### 4.2.3 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

#### 4.2.4 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)

### 4.3 Event storming stages

#### 4.3.1 Big picture

- Extract knowledge about one or several domains
  - Modeling the *current* way of doing business, and *existing* systems
  - Functional *AND* technical stakeholders
- 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
  - Create time marker. 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

#### 4.3.2 Pivotal events and boundaries

- *Pivotal event* is an event that is of particular business importance (and value)
  - Spawns new business processes
  - Involves new stakeholders
  - Commits the business to a financial risk or responsibility
  - Confirms the business receiving a financial benefit
- Can we, along pivotal events, distinguish *Bounded Contexts* in our events?

#### 4.3.3 Process modeling

- Detailed design of one (sub) domain
  - Modeling the *intended* way of doing business, and the *desired* involved systems
  - Functional *AND* technical stakeholders
- Distribute orange, blue, yellow, green and pink post-its
  - Orange: *event*
  - Blue: *command*
  - Yellow (small): *actor (persona)*
  - Pink: *external system, department, time*

  - Green: *read model*
- Remember, a command is in IMPERATIVE, e.g. *Launch missiles Register user subscription request*
- Big picture flows can serve inspiration
- 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)

#### 4.3.4 Aggregates

- What nouns have we discovered that are good candidates for aggregates?
  - Yellow (large): *aggregate*
  - Group the commands by aggregate
- What aggregates would be good candidates for microservices?

# 5 Getting your service used

## 5.1 Public API

### 5.1.1 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?

### 5.1.2 Example API

- Let's look at an example API example API together

  - Its RAML source is available

- Semantic format for describing REST APIs
  - RAML: YAML-based, semantic re-use, easier to write by hand
  - OpenAPI: JSON/YAML-based, more popular

#### 5.1.3 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

### 5.2 Public developer guide

#### 5.2.1 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

#### 5.2.2 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`, ...)

### 5.3 Public service dashboard

#### 5.3.1 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

#### 5.3.2 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

# 6 An introduction to REST

## 6.1 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!

## 6.2 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, ...)

## 6.3 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?

## 6.4 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`, `HEAD`
  - 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)

# 7 A selection of REST patterns

## 7.1 Resource tags and caching

### 7.1.1 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

### 7.1.2 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

### 7.1.3 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
```

## 7.2 Content-type negotiation

### 7.2.1 Resource representation

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

### 7.2.2 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`

### 7.2.3 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/vnd.kamstrup.el.v1+xml
```

## 7.3 Asynchronous and long-running processes

### 7.3.1 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

### 7.3.2 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

### 7.3.3 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

### 7.3.4 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 (*long polling*)

```
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
```

## 7.4 Multi-dimensional versioning

### 7.4.1 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?

### 7.4.2 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

### 7.4.3 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

### 7.4.4 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
```

### 7.4.5 Versioning in query parameters

POST `/users/?quick` { // search 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

#### 7.4.6 Versioning in path

```
GET http://localhost/service/versions/v1/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

#### 7.4.7 Versioning using custom headers

```
GET https://
X-Kamstrup-API-Version: 2022-12-01
```

- 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

# 8 REST API Examples

## 8.1 Examples of REST design

### 8.1.1 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": [ ... ]
    }, [...]
  ]
}
```

### 8.1.2 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

### 8.1.3 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 ...]
```

### 8.1.4 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

### 8.1.5 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>
```

### 8.1.6 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?)

# 9 Micro service life cycle

## 9.1 Dependency management

### 9.1.1 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

### 9.1.2 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

## 9.2 Extending a service

### 9.2.1 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

## 9.3 Testing

### 9.3.1 Unit tests OK, Integration tests not

graphics/tests.mp4

### 9.3.2 Introducing bugs

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

### 9.3.3 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

#### 9.3.4 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

### 9.4 Deployment

#### 9.4.1 Getting your service out there

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

- unknown

#### 9.4.2 Doing deployments

- Automate the environments themselves (`openTF`, `vagrant`, internal DSLs)
- 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)

## 10 Strategy and team dynamics

### 10.1 Succeeding with microservices

#### 10.1.1 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

#### 10.1.2 Migrating your monolith

- Chainsaw anti-pattern
- Strangler pattern
- Modules

#### 10.1.3 Team topologies

- Book by Matthew Skelton and Manuel Pais (2019)
- Very strong resource on team organization, a "modern Conway's law"
- Sketches four kinds of successful teams
  - Stream-aligned team
  - Enabling team
  - Complicated subsystem team
  - Platform team
- And healthy interactions between teams
  - Collaboration
  - X-as-a-service
  - Facilitation

#### 10.1.4 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

## 11 Wrapping up

Thank you for your attention!