

Cloud Computing

Kapitel 2.5: Microservicearchitekturen

Simon Bäumler

simon.baeumler@qaware.de

TH Bingen, 7.11.2017



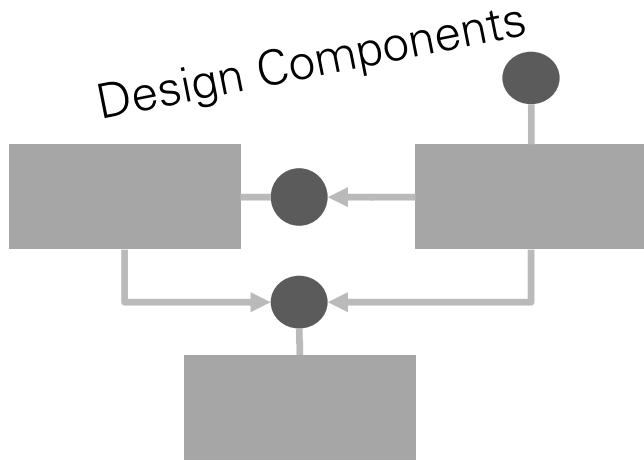
Wieso Microservices?

Klassische Applikationen: Komponenten

DESIGN

BUILD

RUN



1:1

Dev Components



n:1

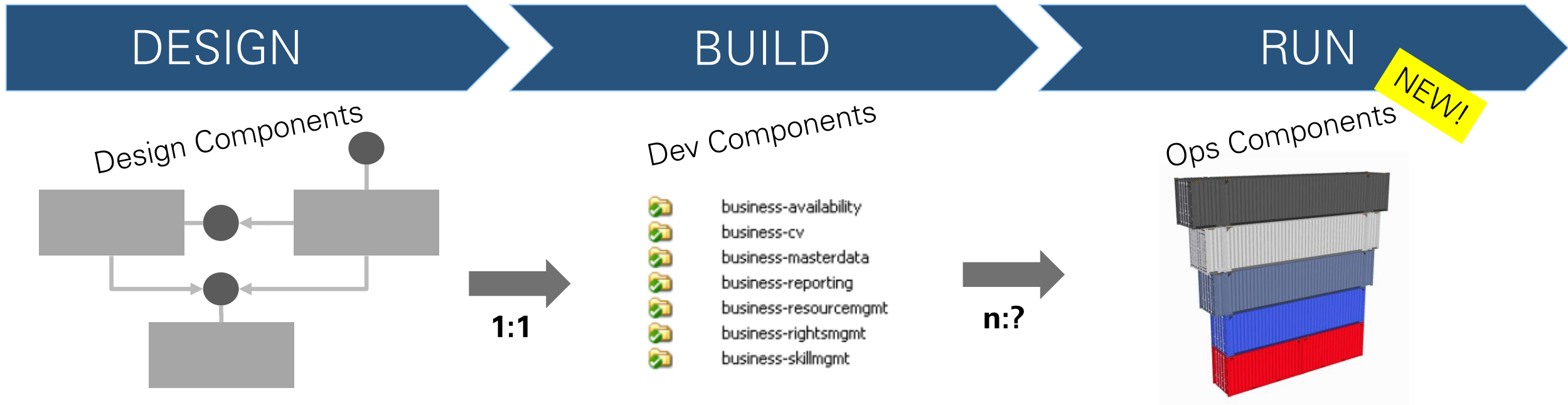
Ops

GlassFish



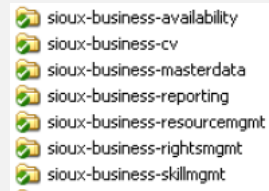
Alle Design/Build Komponenten laufen in einer großen Applikation. Komponentenmodell wird im Betrieb nicht genutzt

Cloud Native Applikationen: Komponenten über den *gesamten* Software Lifecycle



Microservices erlauben den Betrieb einzelner Komponenten zu deployen, zu (re-)starten und zu konfigurieren

Dev Components

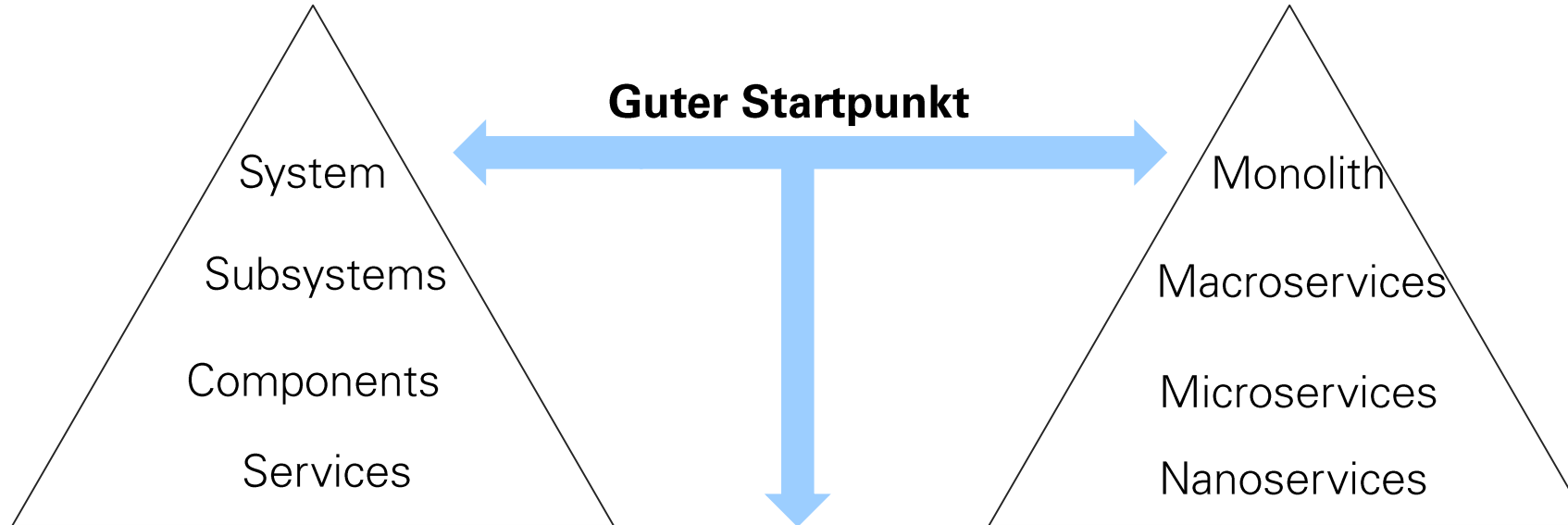


?:1

Ops Components



Guter Startpunkt




+

Trade-offs der Dekomposition

-

- Flexiblere Skalierbarkeit
- Isolation zur Laufzeit (crash, slow-down, ...)
- Unabhängige releases, deployments, teams
- Bessere Auslastung der Ressourcen

- Verteilungsschulden: Latenz
- Erhöht Komplexität der Infrastruktur
- Erhöht Komplexität der Fehlersuche
- Erhöht Komplexität bei Integration



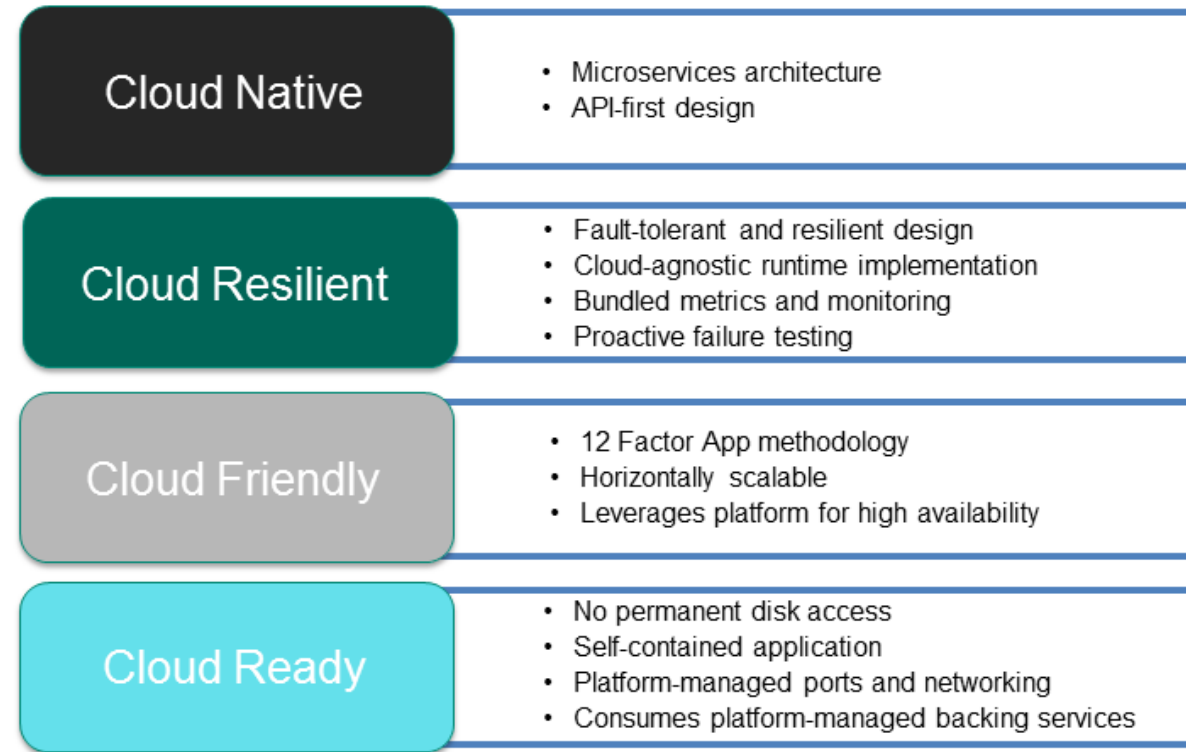
Spring Boot und Spring Cloud: Frameworks für Microservicearchitekturen

Das Spring-Boot Framework erlaubt schnelles Bootstrapping kleiner Applikationen



- Teil des Spring-Ökosystems
 - Spring war ursprünglich ein Dependency Injection Framework, bietet aber heute Lösungen für viele Probleme in der Software Entwicklung.
- „Opinionated Framework“, d.h. häufig benötigte Komponenten (Logging, Monitoring, Konfiguration, ...) sind voreingestellt.
- Spring-Boot ist ein „Convention-over-Configuration“-Framework
 - Reduzierung der notwendigen Entscheidungen beim Aufsetzen des Projekts.
 - Die meisten Designentscheidungen und Konfigurationseinstellungen haben sinnvolle und implizite Defaults.
 - Erst wenn man von diesen Defaults abweicht, muss man als Entwickler tätig werden.

Cloud Native Maturity Model



Source: pivotal.io

Spring-Cloud stellt Komponenten zur Kommunikation zwischen den Services zur Verfügung



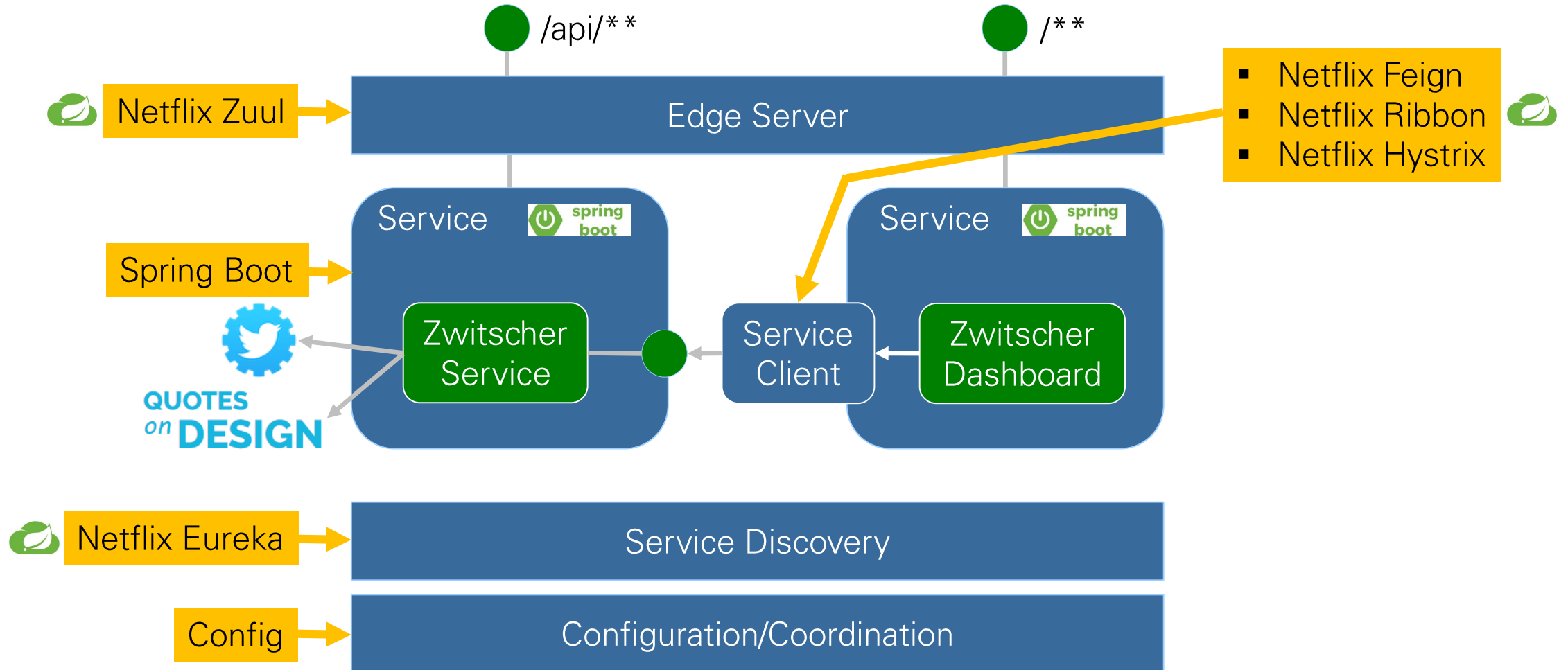
- Netflix Feign / OpenFeign
 - HTTP Rest-Client mit einfacher Konfiguration (siehe Übung)
- Hystix
 - Circuit-Breaker Komponente
 - Einfache Integration in Feign
- Ribbon
 - Loadbalancer
 - Läuft innerhalb des Microservices
 - Einfache Integration in Feign

```
@FeignClient(name = "bookshelf", url = "${services.Bookshelf}")
public interface BookshelfClient {

    /*
     * Define the path for benutzer values
     */
    String VALUE_PATH = "/api/api/books/{isbn}";

    /**
     * Takes the isbn value and returns a book from the bookshelf.
     *
     * @param isbn the isbn number of the book
     * @return the found book
     */
    @ApiOperation(value = "Returns the book object")
    @RequestMapping(value = VALUE_PATH,
        method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
    Book byIsbn(@PathVariable("isbn") String isbn);
}
```


Ein Beispiel einer Spring-Cloud Microservicearchitektur...





Beispiel: Evolution einer Microservicearchitektur

Migration eines Messaging Backbones zu einer Microservice Architektur

Message backend mit Groupware Funktionalität:

- Messaging
- Contacts
- Calendar
- File Store
- etc

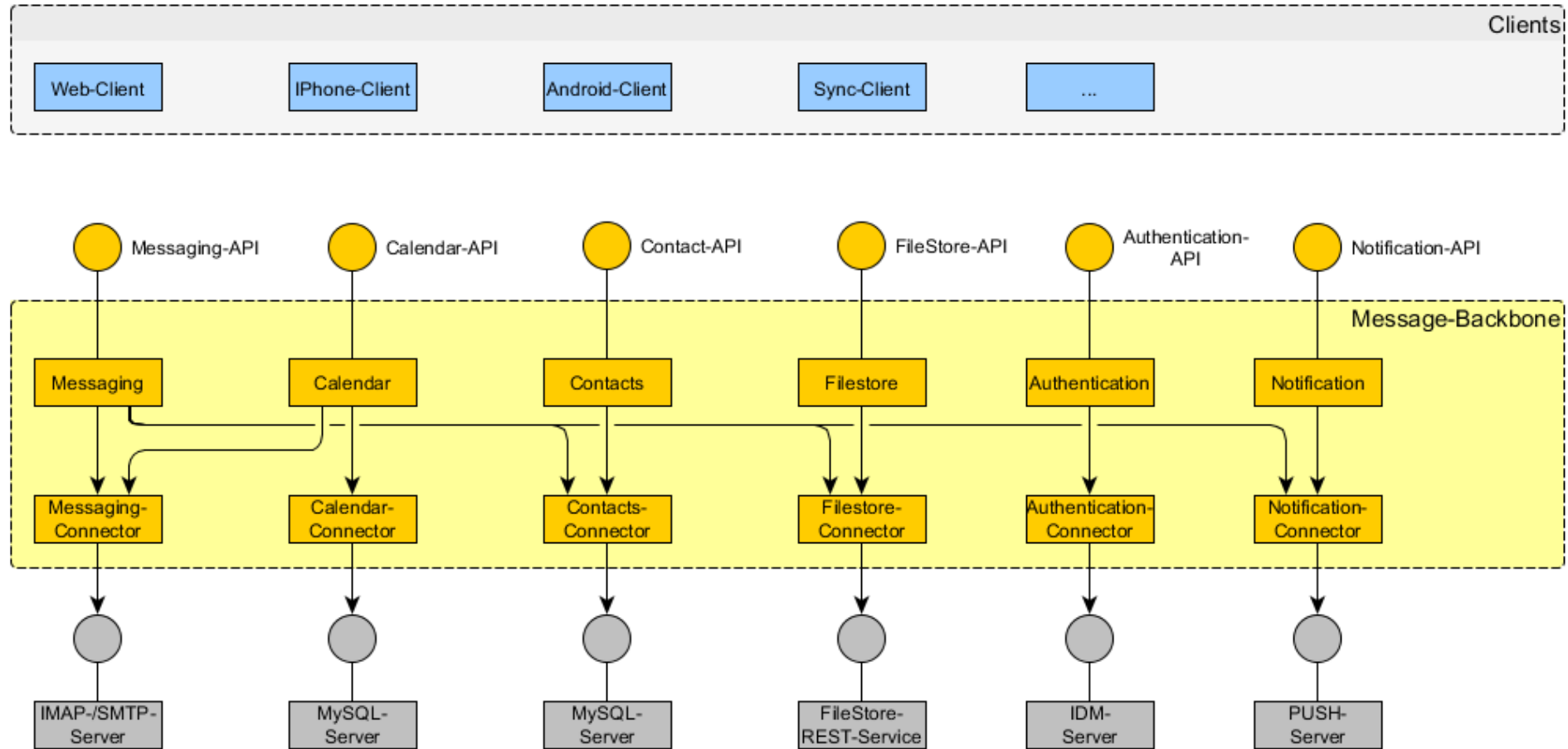
System lief in einem Cluster mit 12 Servern

Codebase: ~ 75k LoC (Java)

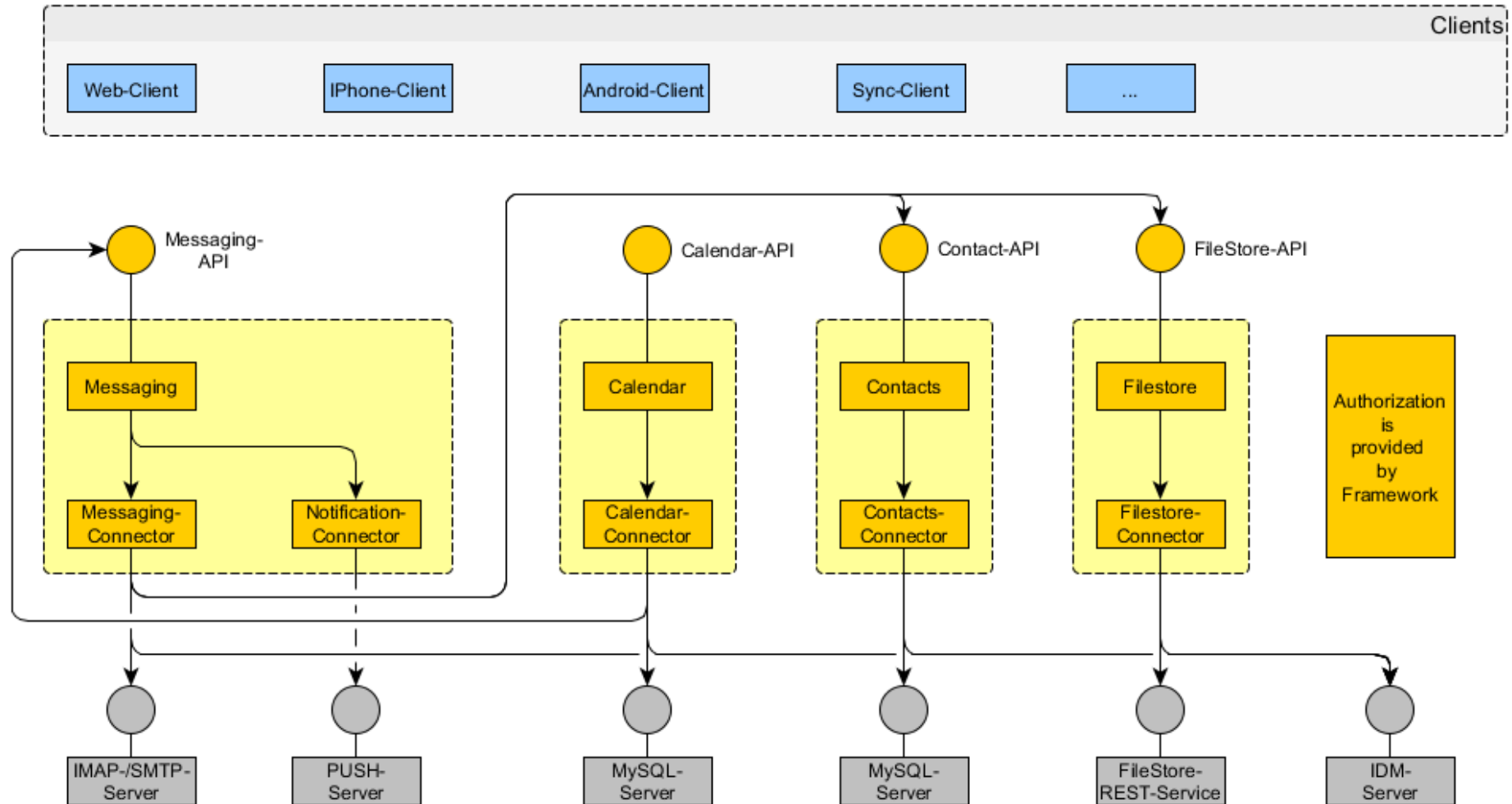
Ca. 4k Requests per second (Das sind >10 Billionen pro Monat)!

Oft ist es legitim (und gut) mit einem Monolithen zu starten!

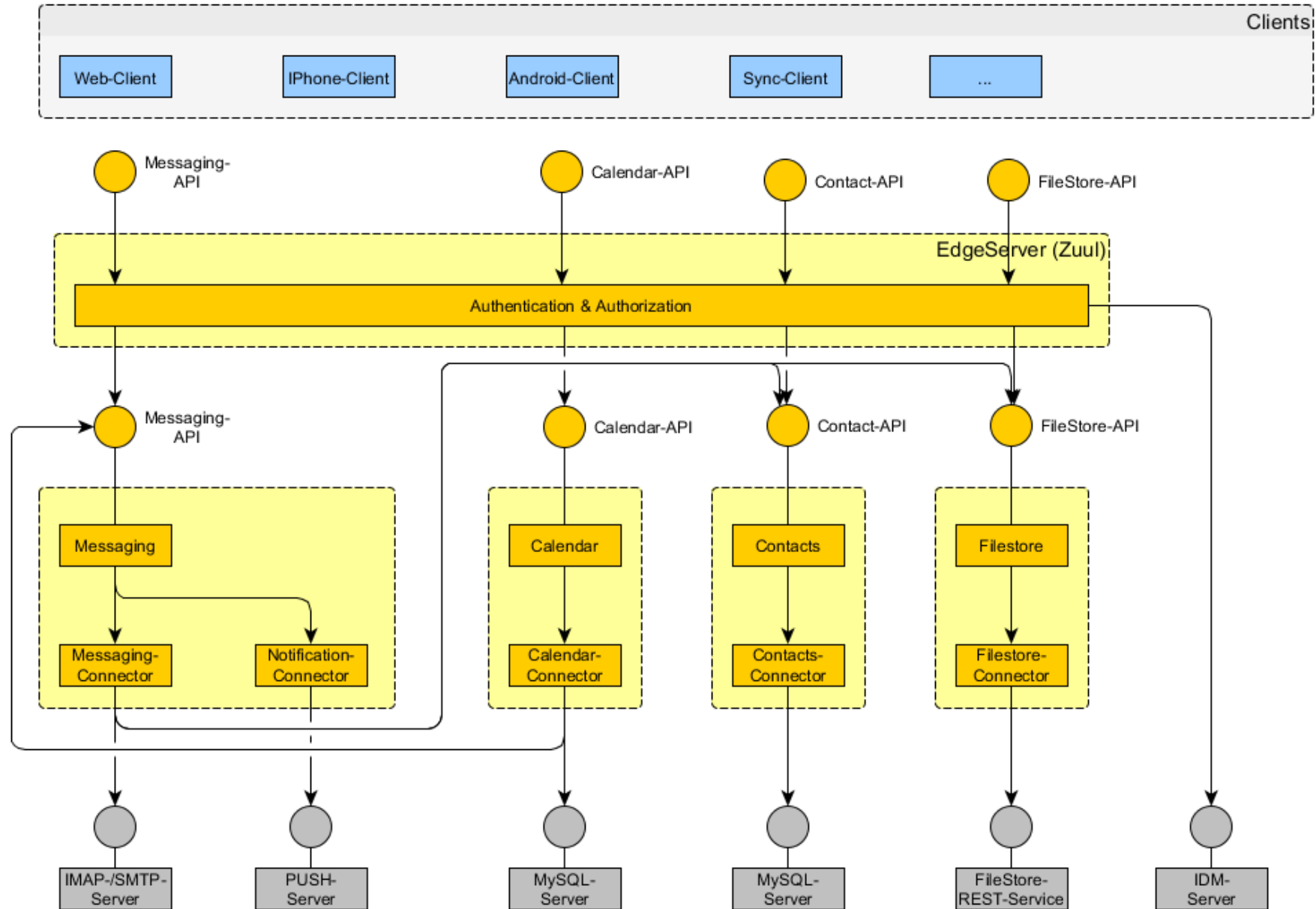
Monolith des Systems vor der Migration.



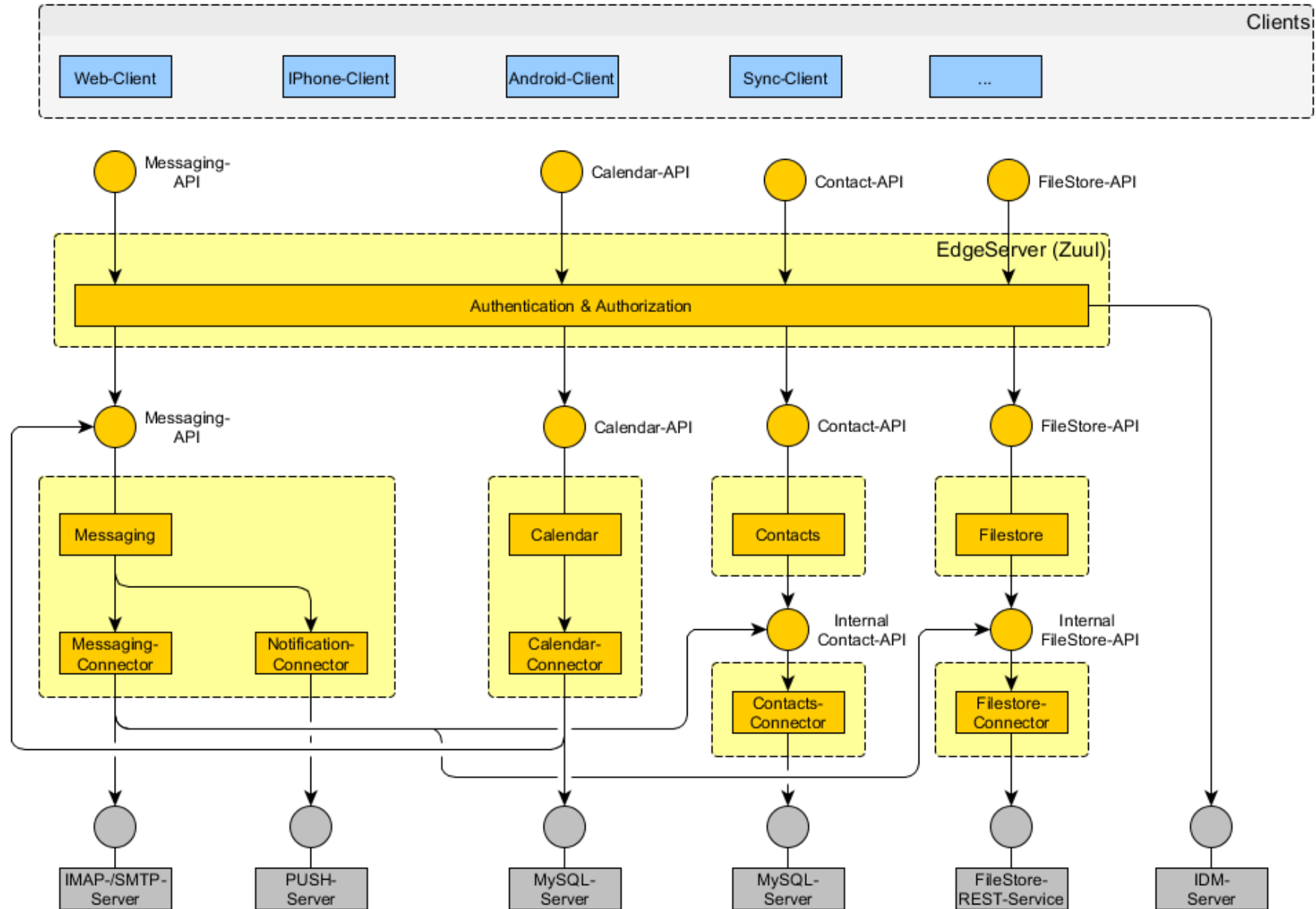
Zerlegung der Semantischen Domains in Einzelkomponenten



Edge-Server hinzugefügt



Weitere dekomposition

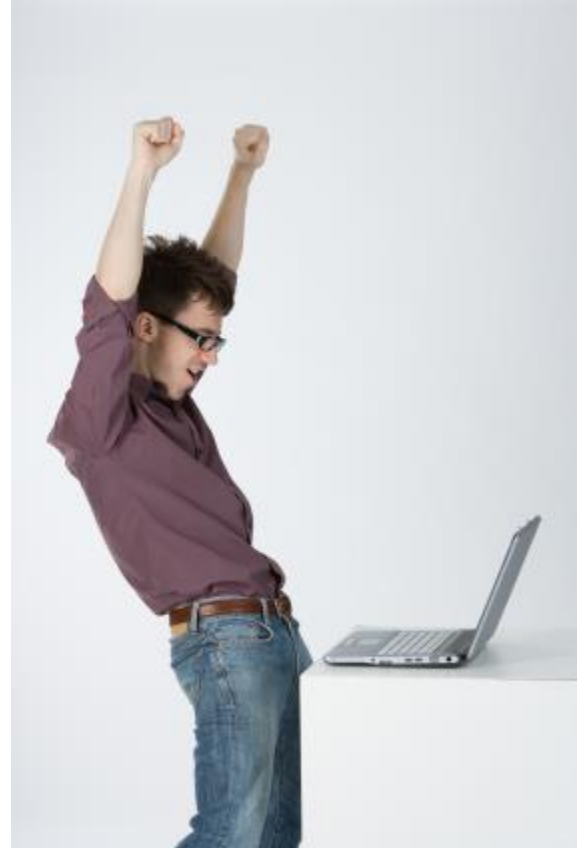




Wichtig für Microservices:
Automatisierung soweit
möglich!

Beschreibung des Builds und des Deployments als Code: Job-DSL Plugins

- Jenkins oder andere Buildplattformen bieten mittlerweile DSLs (Domain Specific Languages) an.
- Mit diesen DSLs kann der Build und das Deployment beschrieben werden (im Unterschied zur manuellen Konfiguration über die UI)
- Das vereinfacht die Pflege der vielen Builds enorm.
- Die Beschreibung ist Teil der Versionskontrolle. Das erleichtert die Nachverfolgung von Änderungen.
- Wiederherstellen oder kopieren von CI-Jobs ist in Sekunden erledigt.

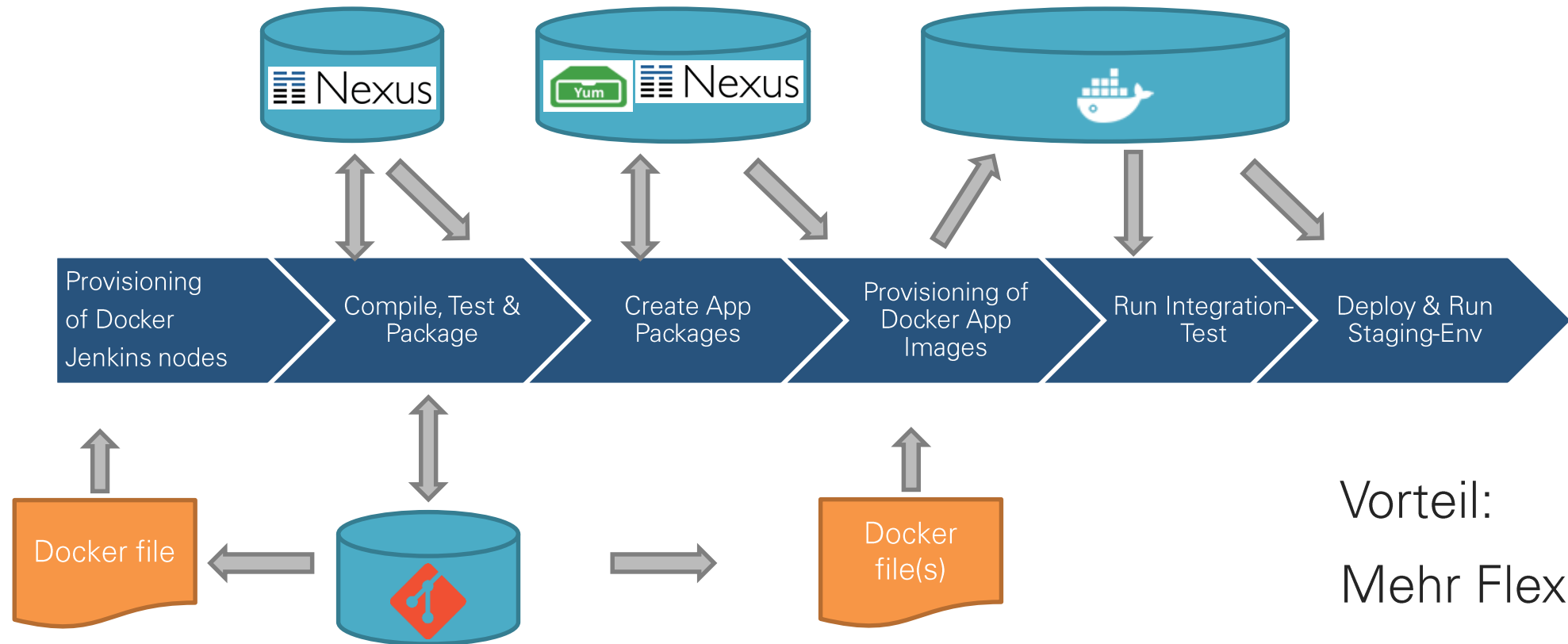


Beispiel:

Continuous-Integration-as-Code mit dem Jenkins Job-DSL plugin

```
job('SAS/SAS-INPUT-QUEUE-BUILD') {  
  
    // additional description of the job  
    description('SAS Input Queue Maven build')  
  
    // configure jdk  
    jdk('jdk-1.8-docker-node')  
  
    // git configuration and trigger  
    scm {  
        git {  
            branch('origin/master')  
            remote {  
                url('https://www.qaware.de/git/SAS')  
                credentials('xxx')  
            }  
            configure { scm ->  
                // configure "git" (not "jgit") and fisheye repository browser  
                scm / gitTool << 'Git'  
                scm / browser(class: ,  
                    hudson.plugins.git.browser.FisheyeGitRepositoryBrowser) {  
                        url('https://www.qaware.de/fisheye/changelog/SAS')  
                    }  
  
                // only include current folder  
                scm / 'extensions' / 'hudson.plugins.git.extensions.impl.PathRestriction' {  
                    'includedRegions'('code/input-queue/.*')  
                }  
            }  
        }  
    }  
    triggers {  
        scm('H/15 * * * *')    // every fifteen minutes (e.g. um :07, :22, :37, :52)  
    }  
  
    // configure docker container to execute maven build  
    wrappers {  
        buildInDocker {  
            dockerHostURI('tcp://nio-build-1.intern.qaware.de:4243')  
            image('10.81.16.196/sas/buildnode')  
            startCommand('/bin/cat')  
        }  
    }  
    configure { node ->  
        // configure the network bridge to 'host'  
        node / buildWrappers  
        / 'com.cloudbees.jenkins.plugins.okidocki.DockerBuildWrapper'  
        / net << 'host'  
    }  
  
    steps {  
        // build dependencies  
        maven {  
            rootPOM('code/commons/pom.xml')  
            goals('clean install -Dmaven.test.failure.ignore=true')  
        }  
        // build input-queue  
        maven {  
            rootPOM('code/input-queue/pom.xml')  
            goals('clean install -Dmaven.test.failure.ignore=true')  
        }  
    }  
  
    // post build publishers  
    publishers {  
        archiveJUnit('**/target/surefire-reports/*.xml')  
    }  
}
```

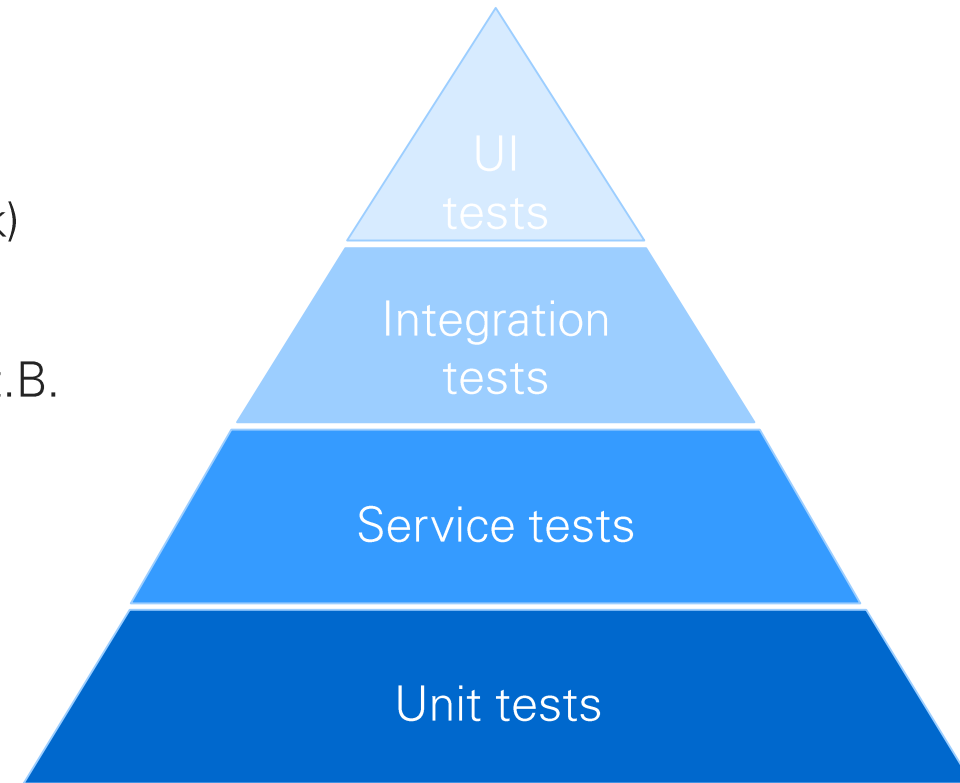
Nutzung Cloud-Nativer Technologien in der CI pipeline: Beispiel Docker-Container



Vorteil:
Mehr Flexibilität
und Durchsatz
im CI Prozess

Aufbau einer Test-Pyramide: Sofortige Feedback bei Fehlern

- Unit Tests: Die klassischen Unit Tests (z.B. JUnit, Mockito)
- Service Tests: Tests eines einzelnen Microservices, inkl. der REST-Controller und Client-Calls (z.B. JUnit, Spring MVC Tests, Wiremock)
 - Mocks der anderen Microservices notwendig (z.B. mit Wiremock)
- Integration Tests:
 - Testet die Integration mehrerer Services und deren Interaktion (z.B. JUnit, Spring MVC Tests)
 - Performance Tests: Testet, ob es signifikante Performance-Änderungen gibt (z.B. Gatling)
- UI-Tests: Testet die UI-Funktionalität und deren Zusammenspiel mit dem Backend (z.B. Selenium, Protractor)

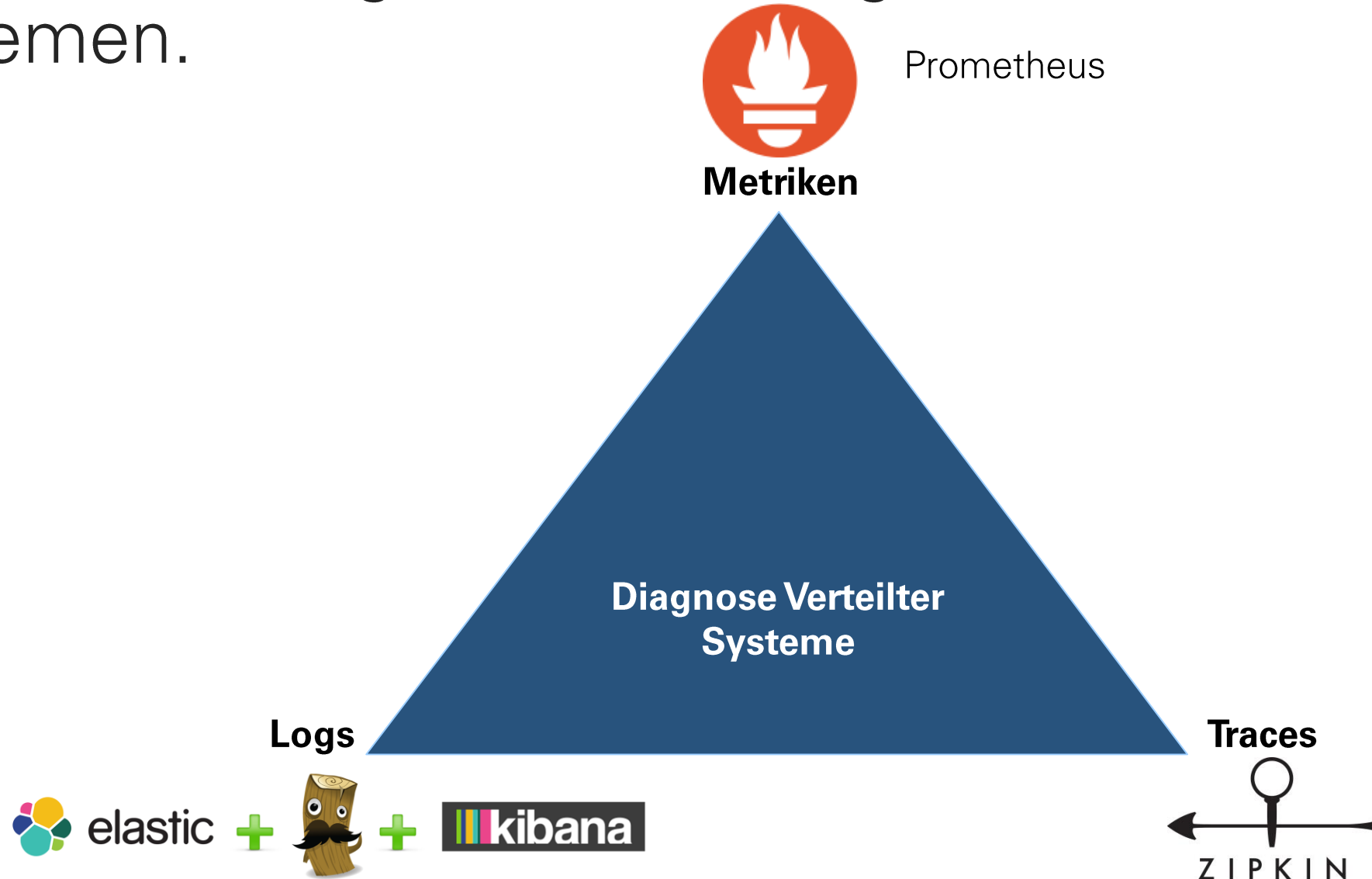


Alle Tests sollten so oft wie möglich ausgeführt werden. Idealerweise bei jedem Commit!

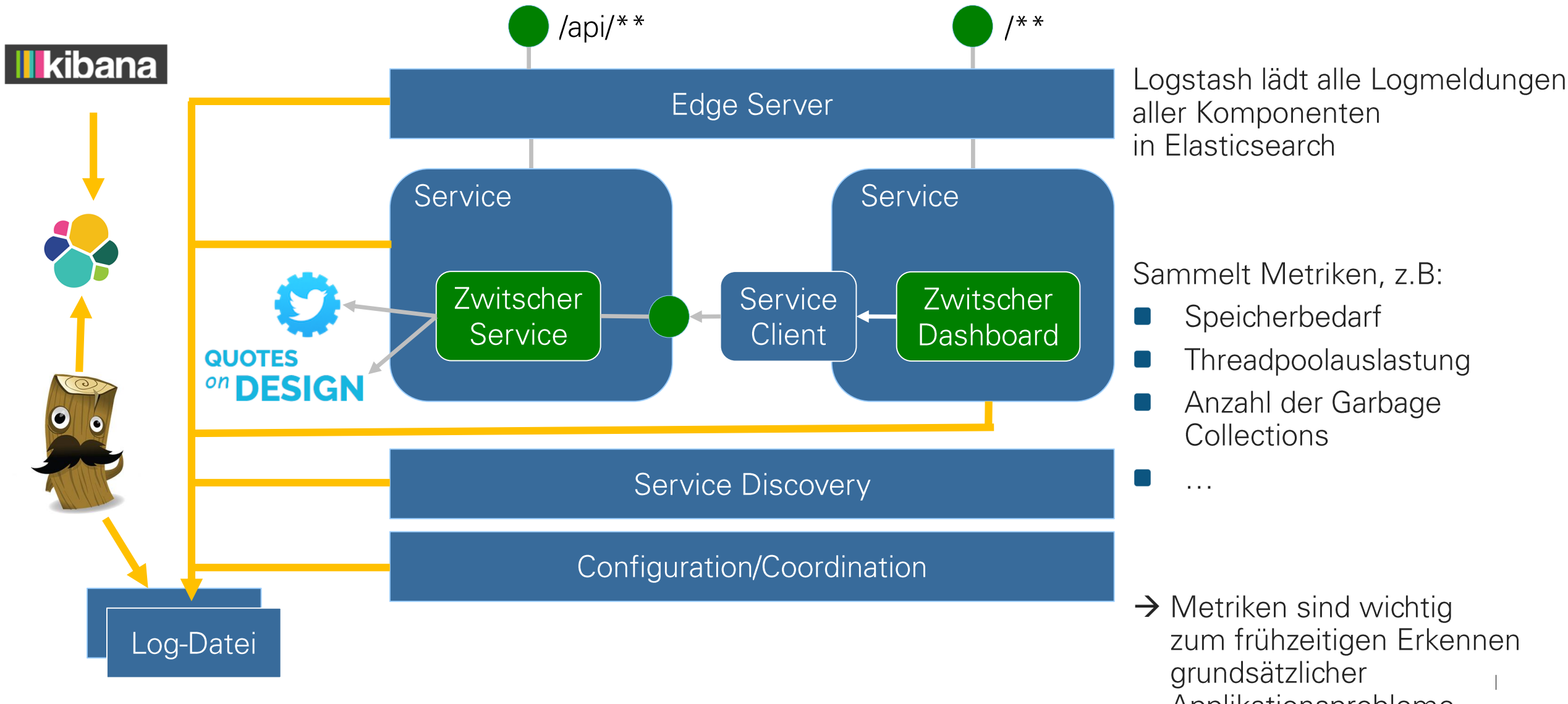


Diagnosability von Microservicearchitekturen

Das „magische“ Diagnosedreieck als Antwort auf die Herausforderungen bei der Diagnose von Verteilten Systemen.



ELK konsolidiert, indiziert und visualisiert die Log-Einträge aller Komponenten.

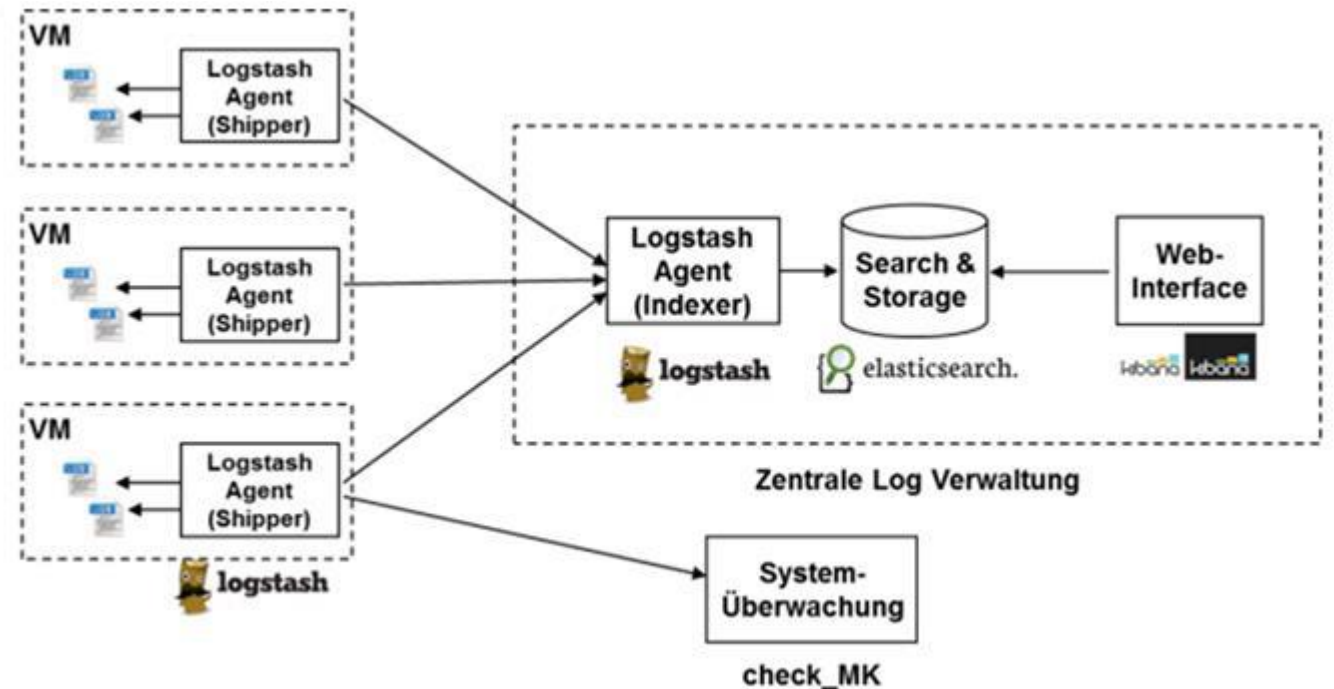


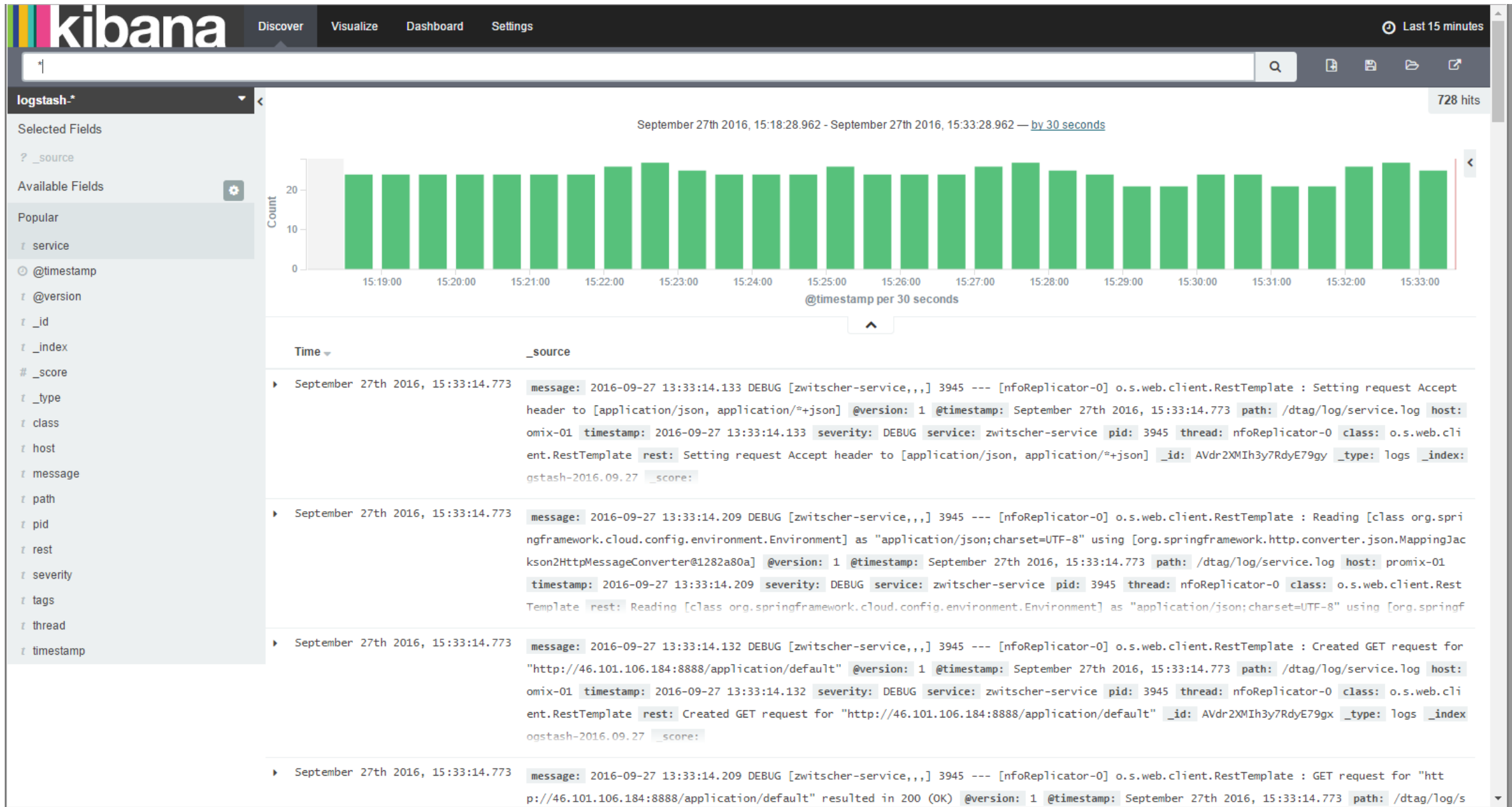
Mittlerweile weit verbreitet: ELK – **E**lasticSearch, **L**ogstash und **K**ibana.

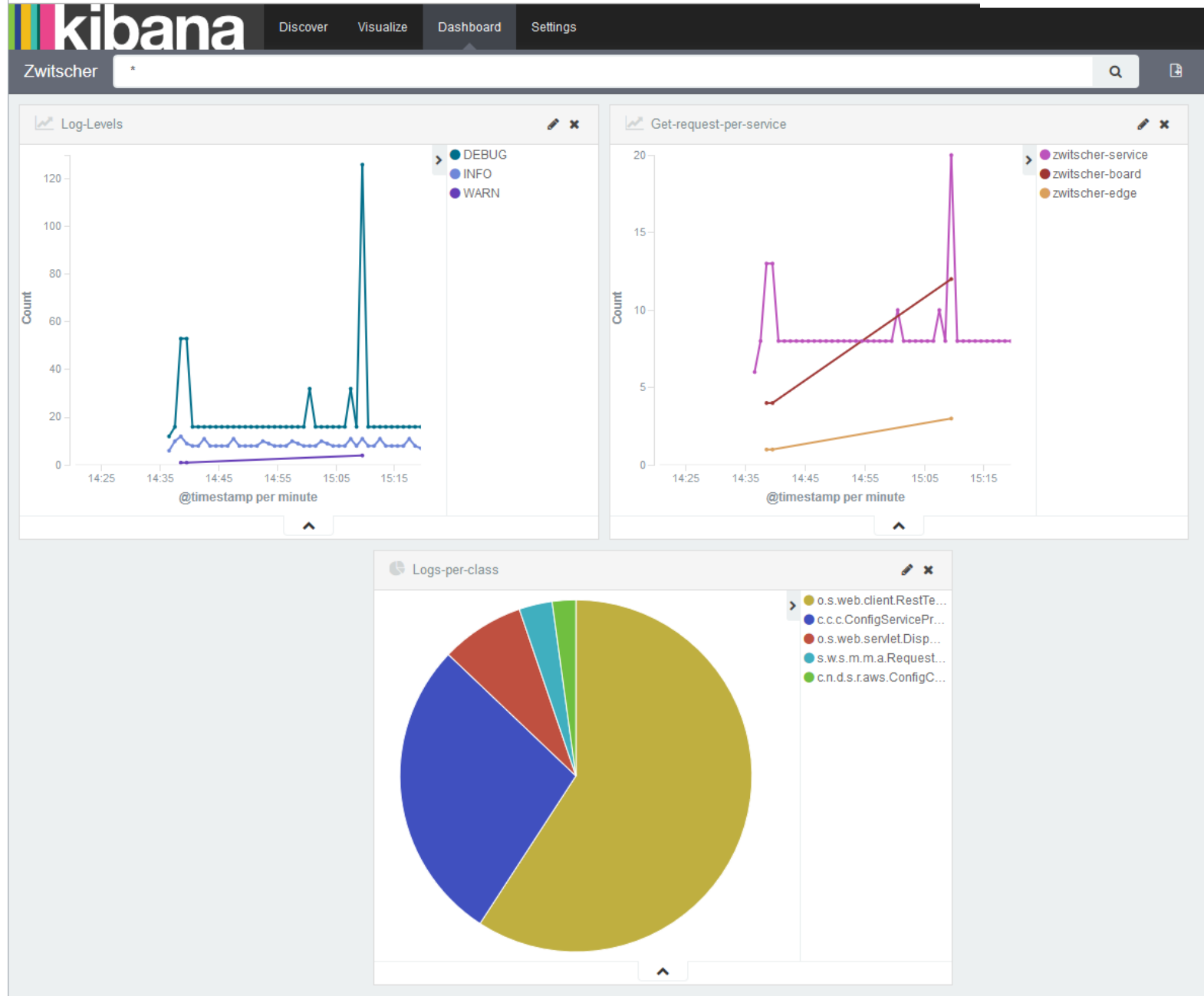


Weitverbreiteter Baustein zum Sammeln von Log-Dateien.

- **Logstash** ist für sammeln und verarbeiten der Logs zuständig.
- **ElasticSearch** ist eine Suchmaschine / NoSQL Datenbank, die als zentraler Speicher dient und Volltextsuche auf den gespeicherten Log-Einträgen ermöglicht.
- **Kibana** dient zur Visualisierung der Anfragen an ElasticSearch.



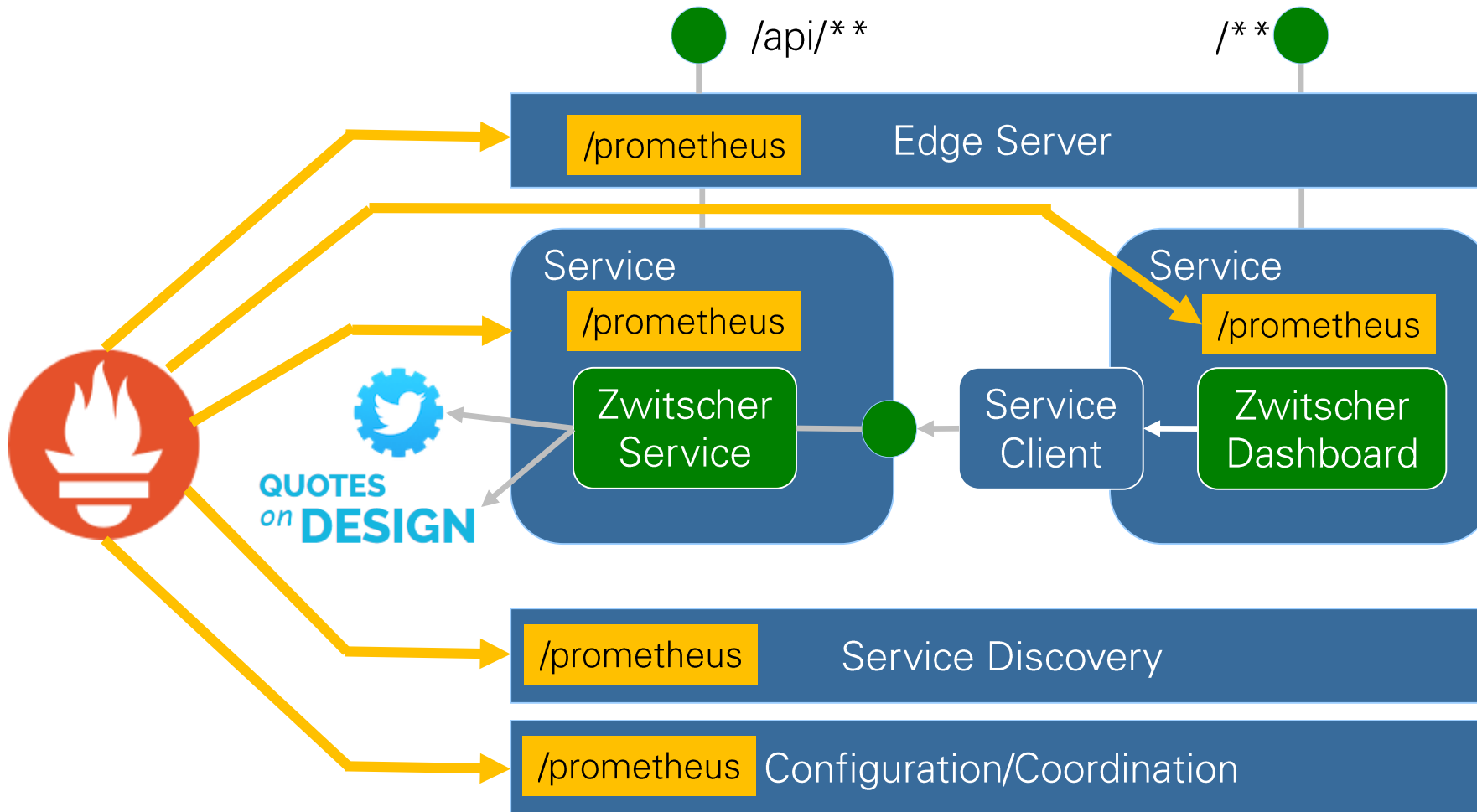




... ausgestattet mit Prometheus zur kontinuierliche Kontrolle der Metriken.



Servlet



Sammelt Metriken, z.B:

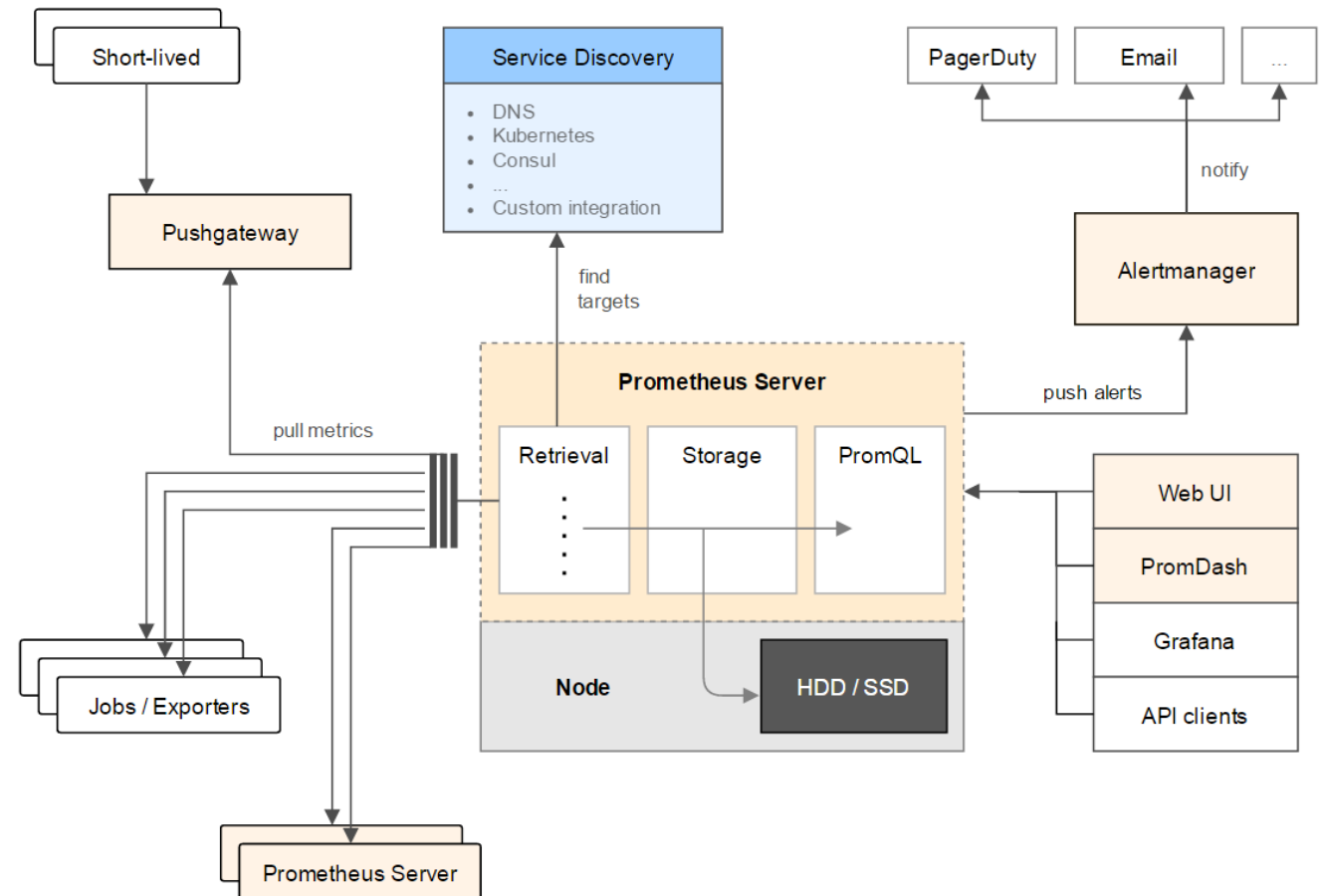
- Speicherbedarf
- Threadpoolauslastung
- Anzahl der Garbage Collections
- ...

→ Metriken sind wichtig zum frühzeitigen Erkennen grundsätzlicher Applikationsprobleme (z.B. zu wenig Speicher)

Prometheus speichert die Laufzeitinformationen von Cloud Native Anwendungen.



- Prometheus = Sammeln von Laufzeitinformationen (Metriken) + Speicher von Laufzeitinformationen + Alerting
- Architekturablauf (Grob):
 - Anwendungen müssen im Vorfeld instrumentiert werden (Unzählige Bibliotheken sind verfügbar).
 - Der Prometheus Server holt die Metriken ab.
 - Die Daten werden typischerweise 14 Tage lang gespeichert.
- Zur längeren Speicherung benötigt man einen Long-Term-Storage für Zeitreihen.
 - Chronix, InfluxDB, OpenTSDB, ...





Targets

Board				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:8081/prometheus	UP	none	4.1s ago	
Config				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:8888/prometheus	UP	none	272ms ago	
Edge				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:8765/prometheus	UP	none	2.225s ago	
Eureka				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:8761/prometheus	UP	none	404ms ago	
prometheus				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	none	2.644s ago	

Load time: 73ms
Resolution: 86s

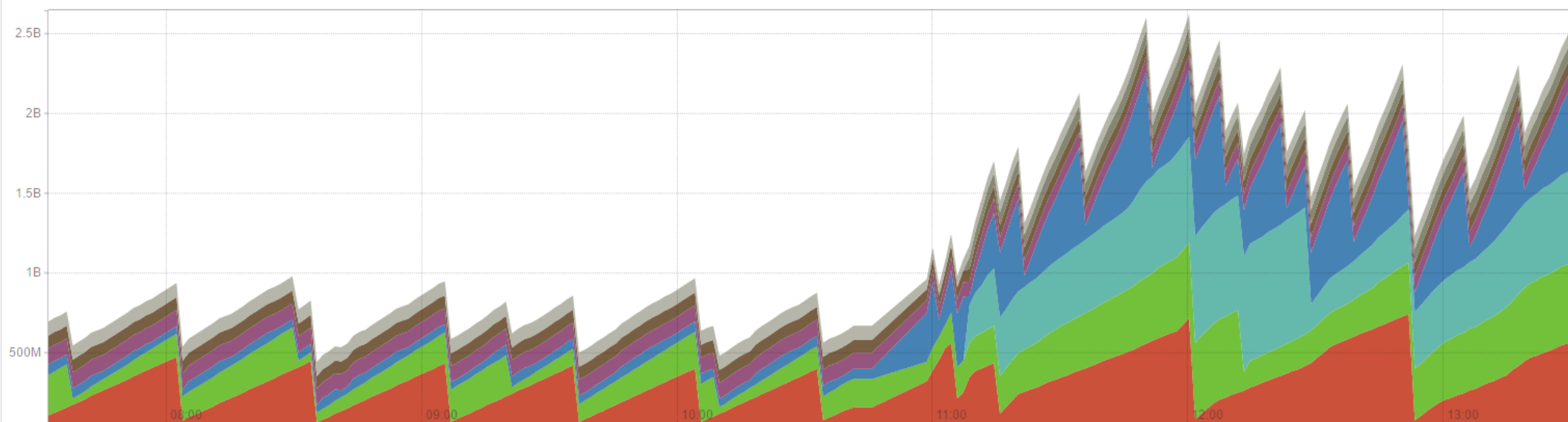
jvm_memory_bytes_used

Execute

jvm_memory_bytes_used ▾

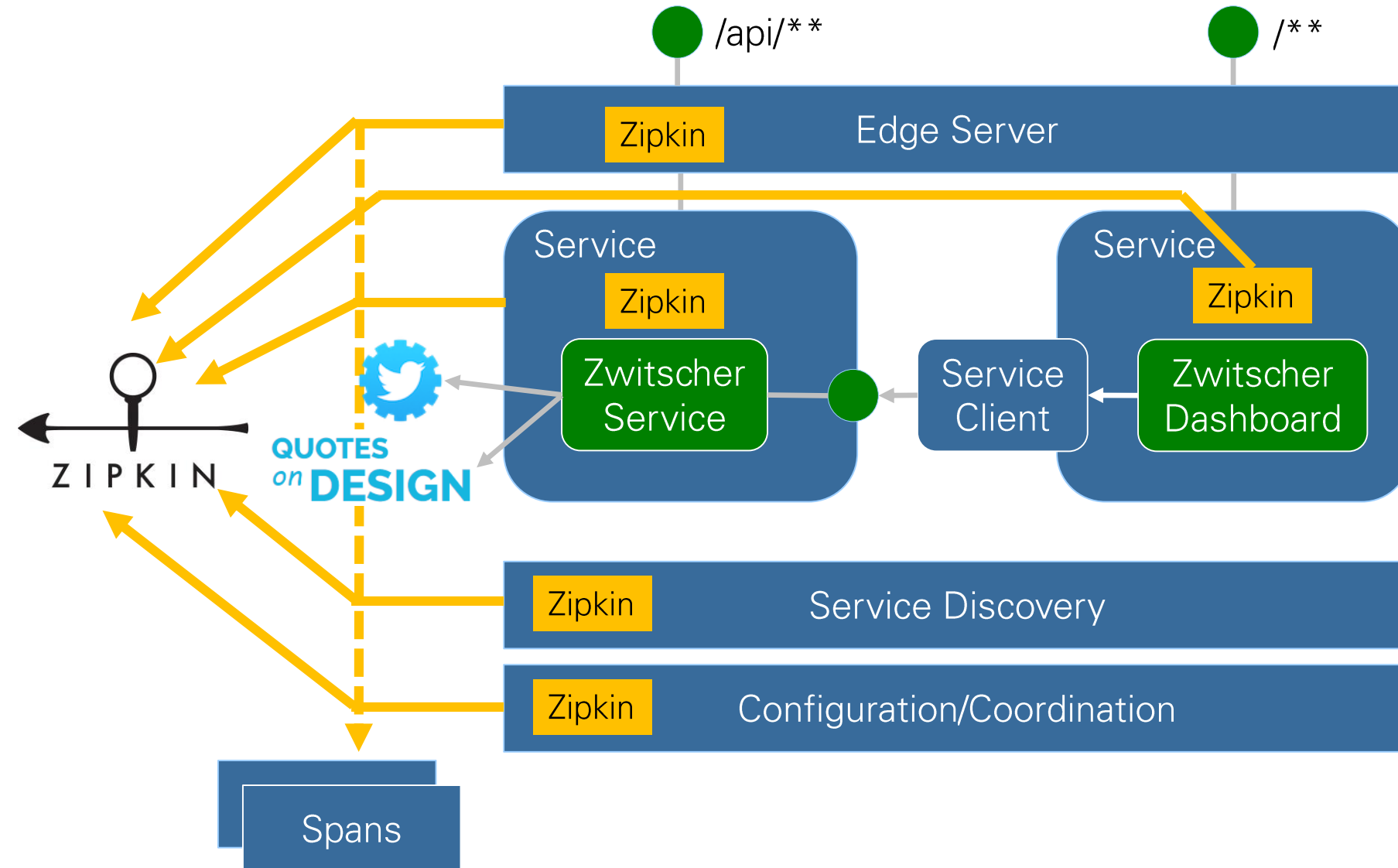
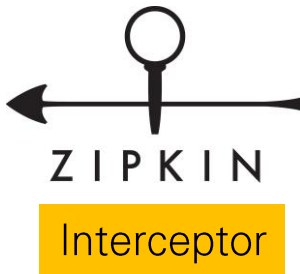
Graph Console

- 6h + ⏪ Until ⏩ Res. (s) stacked



```
jvm_memory_bytes_used{area="nonheap",instance="localhost8888",job="Config"}
jvm_memory_bytes_used{area="nonheap",instance="localhost8765",job="Edge"}
jvm_memory_bytes_used{area="nonheap",instance="localhost8761",job="Eureka"}
jvm_memory_bytes_used{area="nonheap",instance="localhost8081",job="Board"}
jvm_memory_bytes_used{area="heap",instance="localhost8888",job="Config"}
jvm_memory_bytes_used{area="heap",instance="localhost8765",job="Edge"}
jvm_memory_bytes_used{area="heap",instance="localhost8761",job="Eureka"}
jvm_memory_bytes_used{area="heap",instance="localhost8081",job="Board"}
```

... ausgestattet mit Zipkin, zur Nachverfolgung was passiert und wo es passiert und wie es passiert.



Die Architektur von Zipkin ist einfach. Jeder Service schickt seine Tracing-Informationen.

Anwendungen müssen mit Zipkin-Bibliotheken instrumentiert sein.

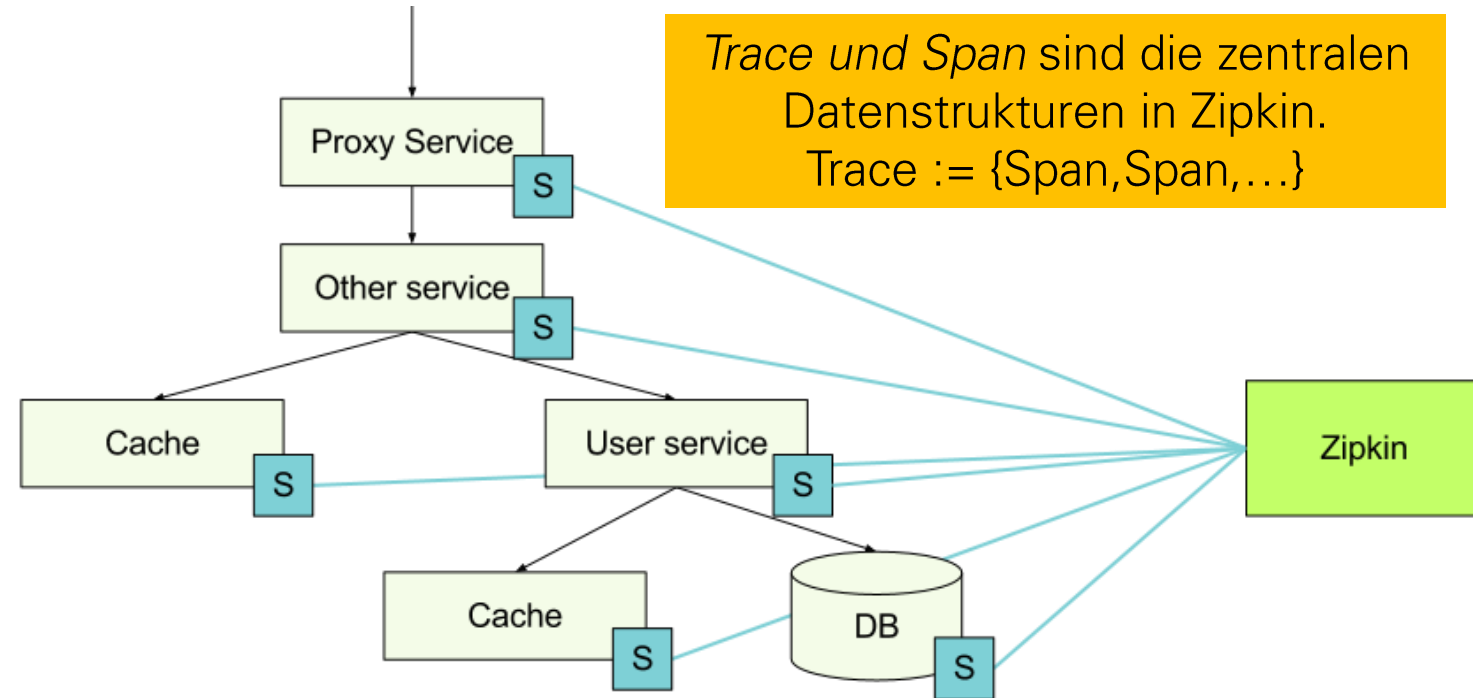
- Es gibt zahlreiche Bibliotheken.

Die Zipkin-Bibliotheken reichern die Requests mit Tracing-Informationen an.

- Aber auch: Spezifische Informationen

Zipkin baut aus diesen Informationen einen Trace zusammen.

- Keine Abhängigkeit zwischen den Services



Zipkin basiert auf Google's Dapper Papier: <http://research.google.com/pubs/pub36356.html>

Zipkin verarbeitet die Traces mit einer etablierten Werkzeugkette von Open-Source Bausteinen.



Transportmöglichkeiten

- HTTP, Kafka, **Log** oder Scribe

Collector

- Validiert, speichert und indiziert Spans.

Storage

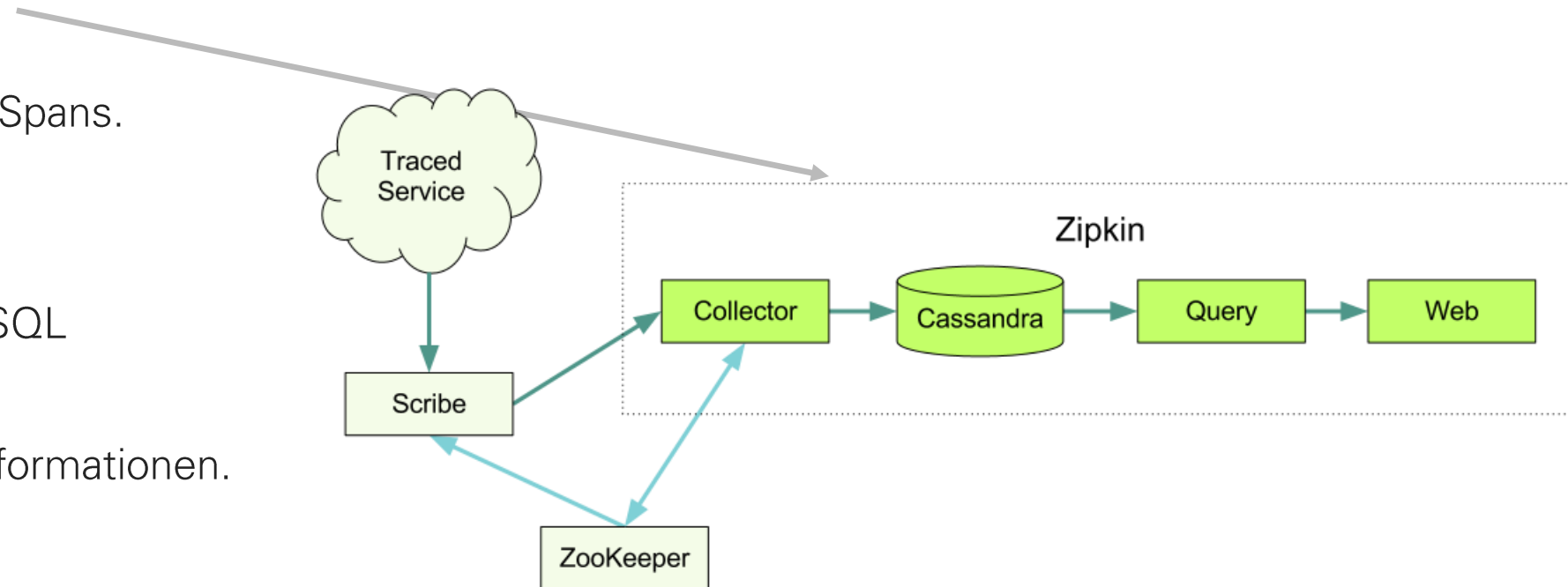
- Initial für Cassandra entworfen.
- Alternativen: ElasticSearch, MySQL

Zipkin Query Service

- JSON API zum Abgreifen der Informationen.

Web UI

- Visualisieren der Daten.



Duration: 2.028s Services: 3 Depth: 6 Total Spans: 8

JSON

Expand All Collapse All Filter Service S...

zwitscher-board x5 zwitscher-edge x2 zwitscher-service x4

Services		405.600ms	811.200ms	1.217s	1.622s	2.028s
+ zwitscher-edge	2.028s : http:/search	-	-	-	-	-
- zwitscher-edge	2.021s : http:/search	-	-	-	-	-
- zwitscher-board	2.004s : http:/search	-	-	-	-	-
- zwitscher-board	1.985s : user-search	-	-	-	-	-
- zwitscher-board	404.000ms : http:/quote	-	-	-	-	-
zwitscher-service	394.000ms : http:/quote	-	-	-	-	-
- zwitscher-board	-	-	-	-	566.000ms : http:/tweets	-
zwitscher-service	-	-	-	-	560.000ms : http:/tweets	-

Zipkin Investigate system behavior Find a trace Dependencies

Duration: 2.028s

Expand All Collapse All

zwitscher-board x5 zwits

Services

- + zwitscher-edge 2.0
- zwitscher-edge 2.0
- zwitscher-board 2.0
- zwitscher-board 2.0
- zwitscher-board 2.0
- zwitscher-service 2.0
- zwitscher-board 2.0
- zwitscher-service 2.0

zwitscher-board.user-search: 1.985s

AKA: zwitscher-board

Date Time	Relative Time	Annotation	Address
27.9.2016, 14:39:53	429.000ms	quote-loaded	46.101.106.184:8081 (zwitscher-board)
27.9.2016, 14:39:55	2.004s	tweets-loaded	46.101.106.184:8081 (zwitscher-board)

Key	Value
Local Component	unknown
search-query	Oktoberfest
Local Address	46.101.106.184:8081 (zwitscher-board)

Go to trace

JSON

2.028s



Bonusmaterial

Module structure of a service: We always create a client module with the API

```
package sas.service.a.api;

public interface ServiceAPI {
    @RequestMapping(value = "service/path",
        produces = MediaType.APPLICATION_JSON_VALUE,
        method = RequestMethod.GET)
    ResultDTO restServiceMethod(@PathVariable("id") String id) ;
}

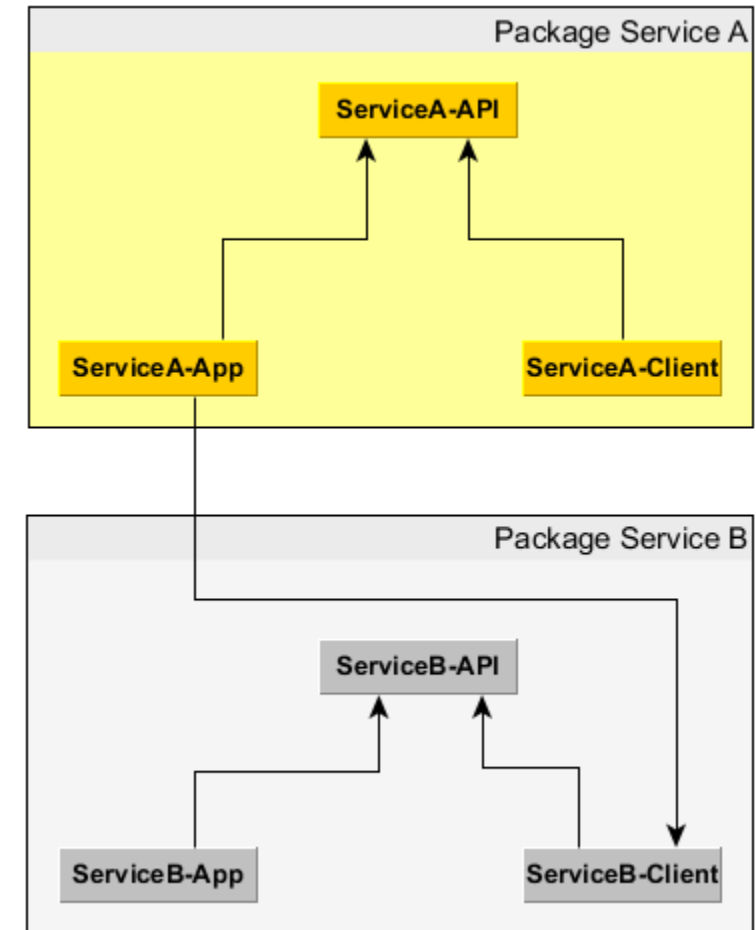
package sas.service.a.app;

@RestController
public class ServiceController implements ServiceAPI {
    @Override
    public ResultDTO restServiceMethod(@PathVariable("id") String id) {
        // implement service here
    }
}

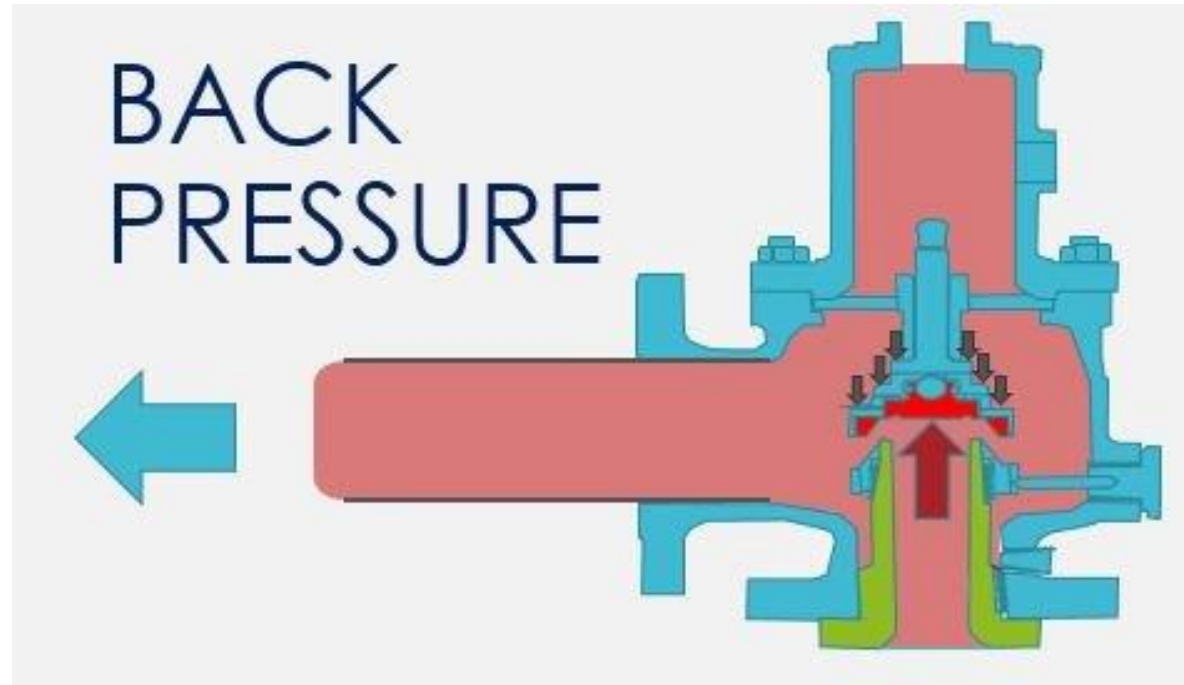
package sas.service.a.client;

@FeignClient(url = "${services.serviceurl}")
public interface ServiceClient extends ServiceAPI {
    // no implementation is needed, as Netflix Feign takes care of that
}
```

Runnable code and configuration can be created by a Maven Archetype



Besides horizontal scaling, elasticity means intelligent handling of excessive loads





Inflow of data and requests

- Usually not constant (low tide, high tide)
- Unexpected variation may occur (flood, drought)

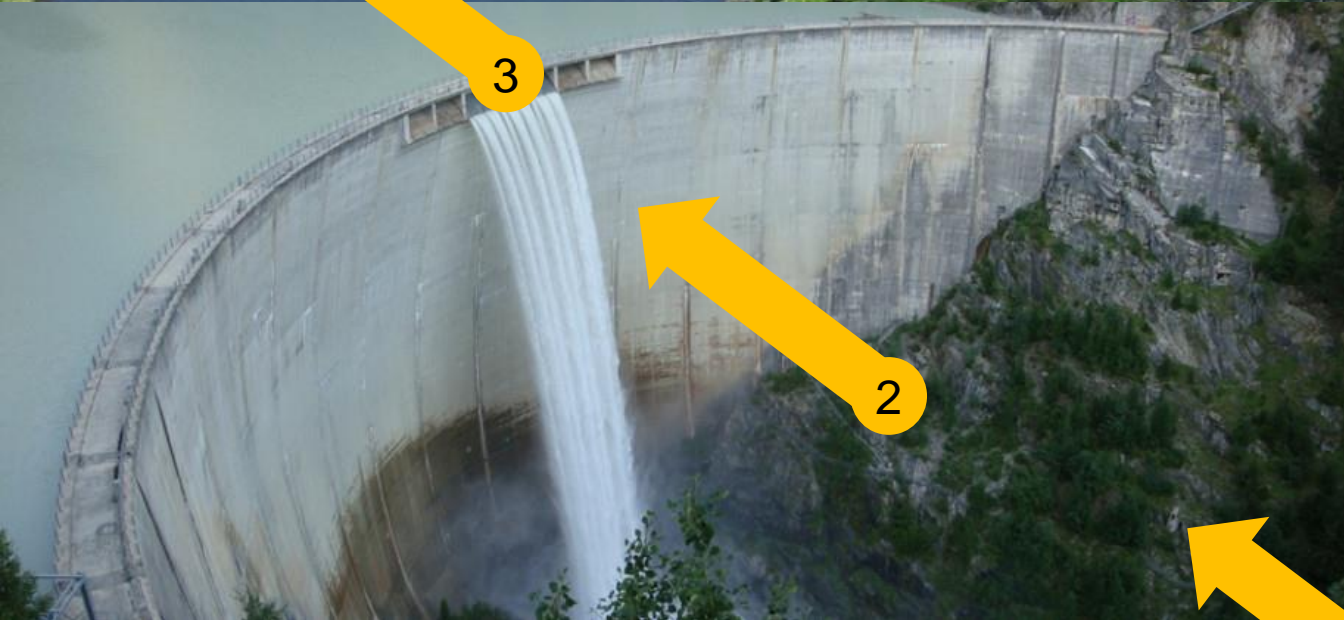


?



Processing of data and requests

- Maximum rate, that can be processed without problems
- Rate, where the system is damaged



3

Dam up in a big dam lake

2

Adjusts the valve, so that the actual max. flow rate is not exceeded

1

Reports, how much max. flow is currently possible

