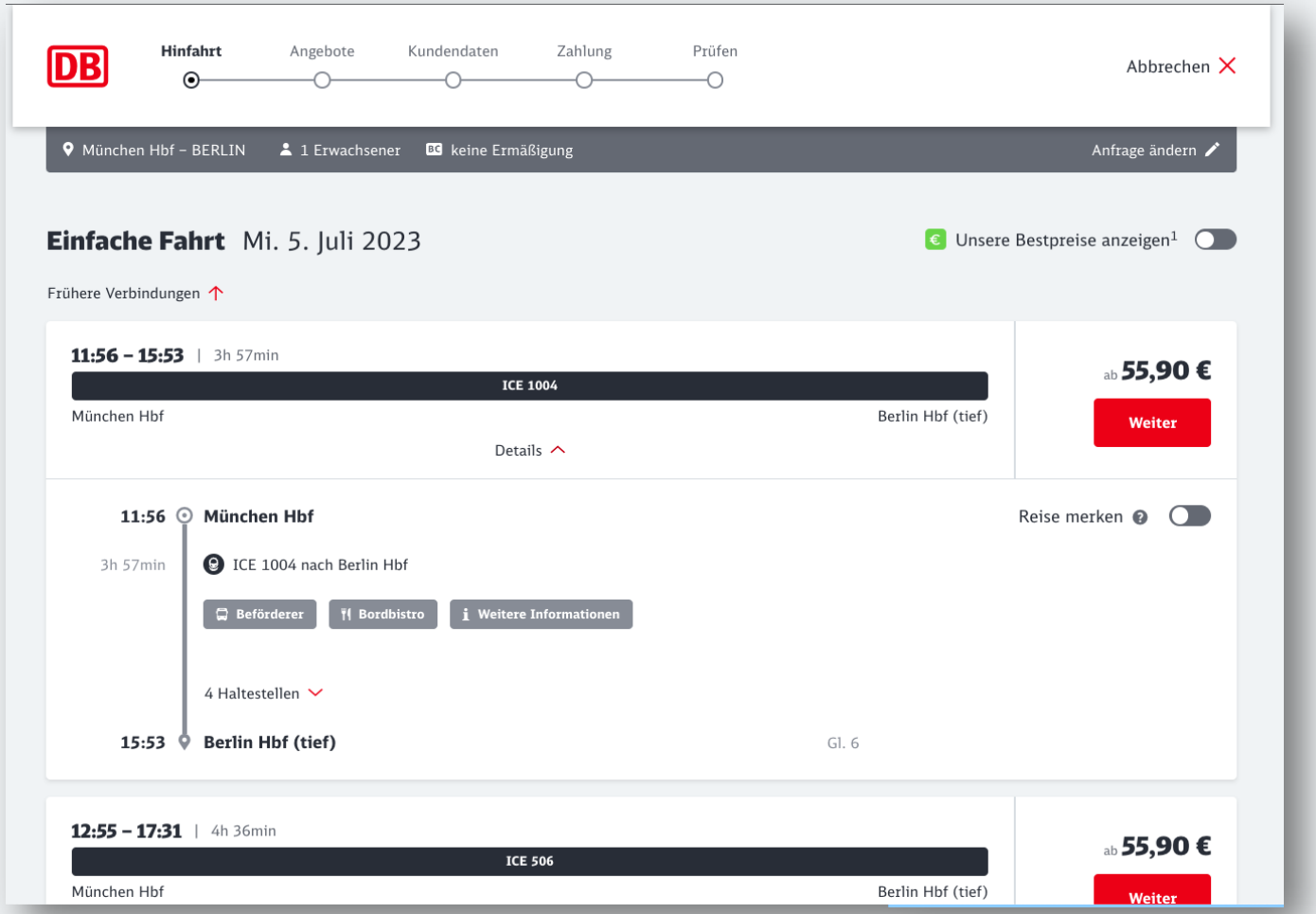
Performance-Optimierung

Wie wir mit bahn.de über 2.000 req/s beantworten

Heiko Maaß – System Engineer @ DB Fernverkehr AG
Lukas Pradel – System Engineer @ DB Fernverkehr AG

31.7.2024 |

Relaunch bahn.de / Navigator 2023



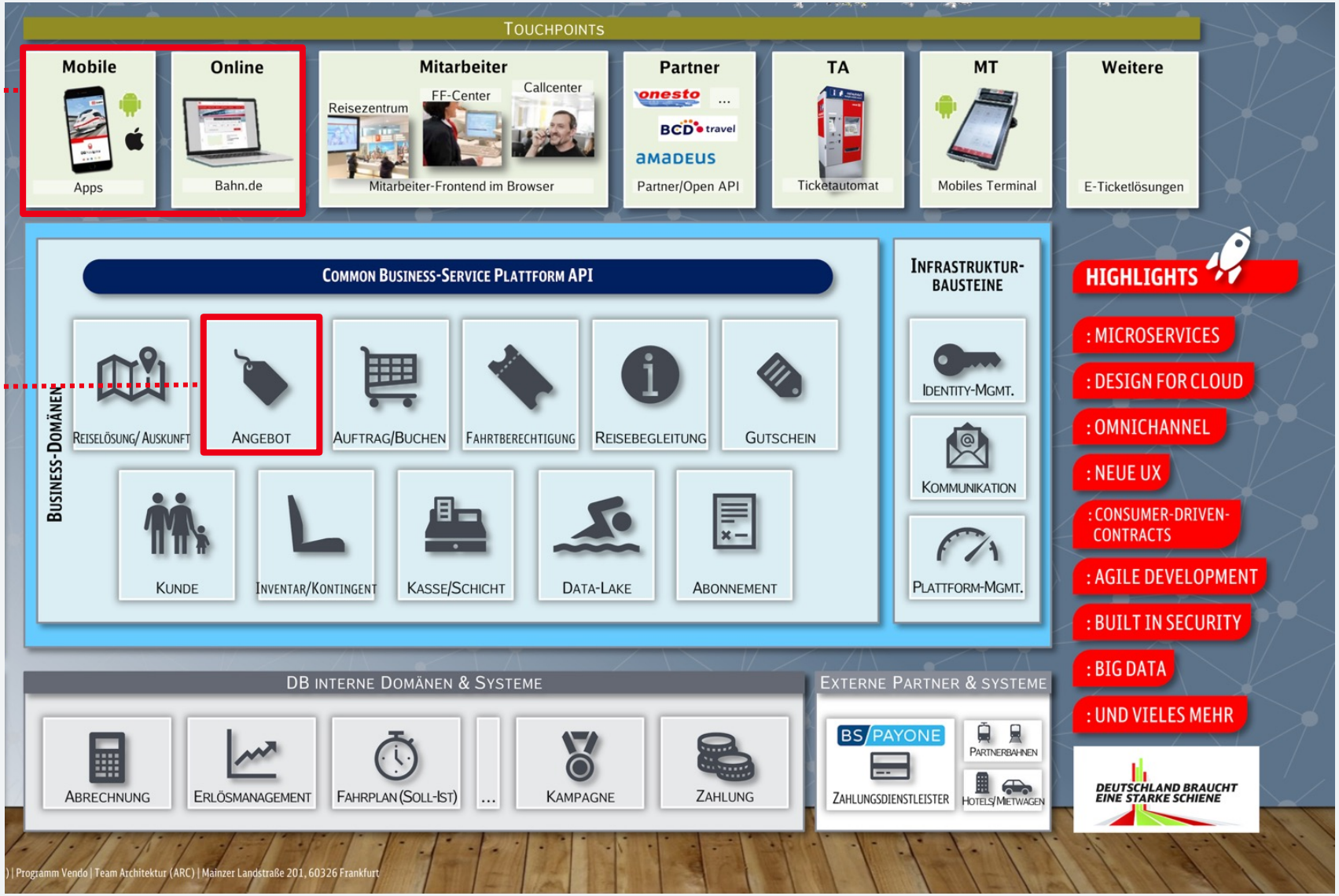
bahn.de

DB Navigator

Programm Vendo 2017 – 2023

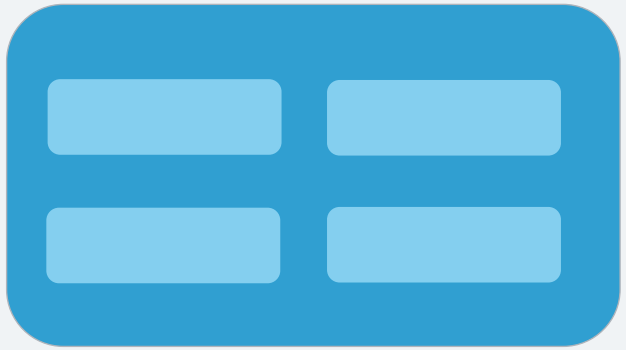
Sichtbar für den Kunden

Unser Fokus heute



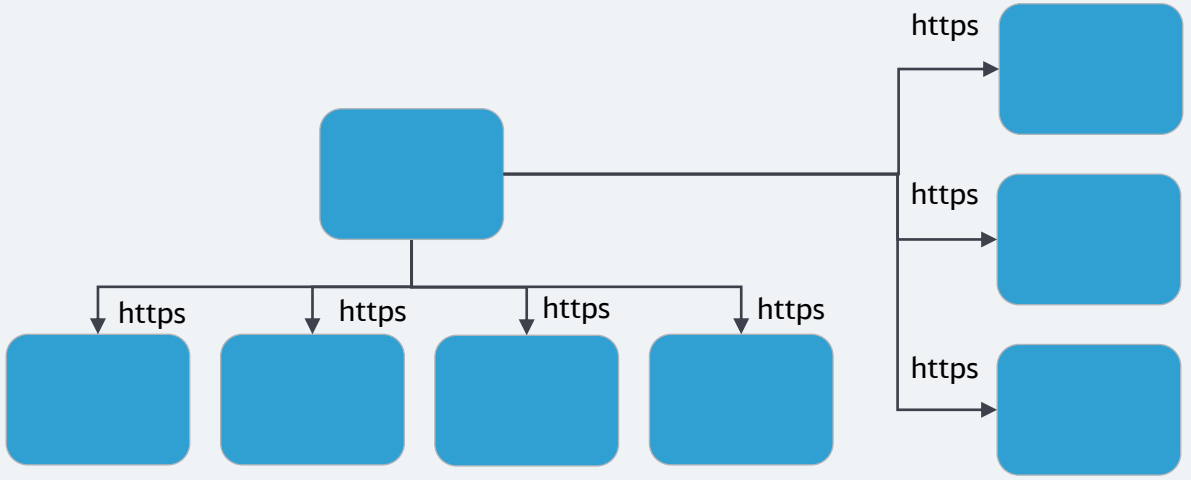
Ausgangslage Angebotserstellung 2017

Bestandssystem



- C/C++ basiert
- Modularisiert mit Bibliotheken
- Binäre Interprozess-Kommunikation
- Abgebotstemplates in Memory

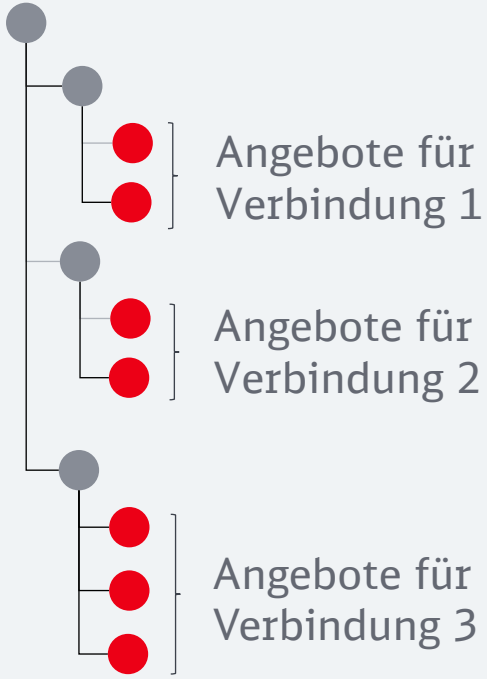
Neue Architektur

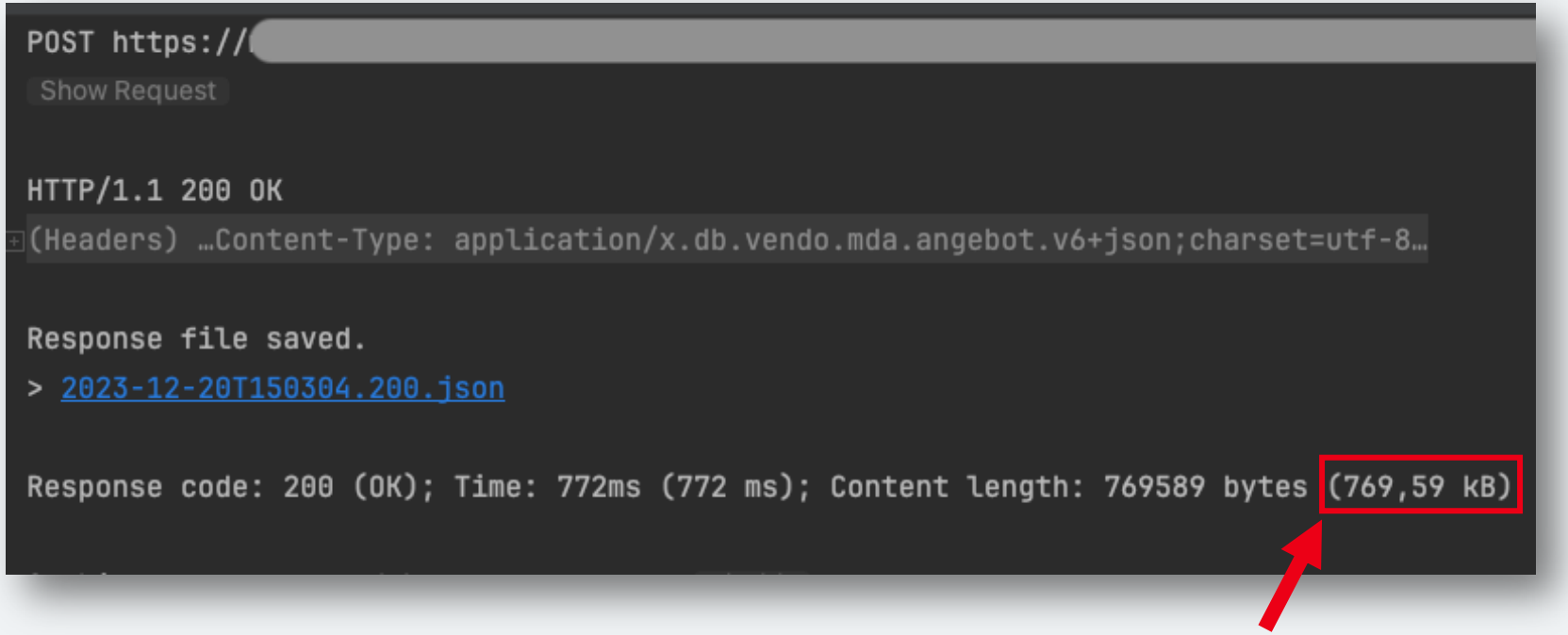


- Java-basiert
- Modularisiert mit „Micro“-services
- Kommunikation via HTTP/JSON
- Angebotstemplates in Memory

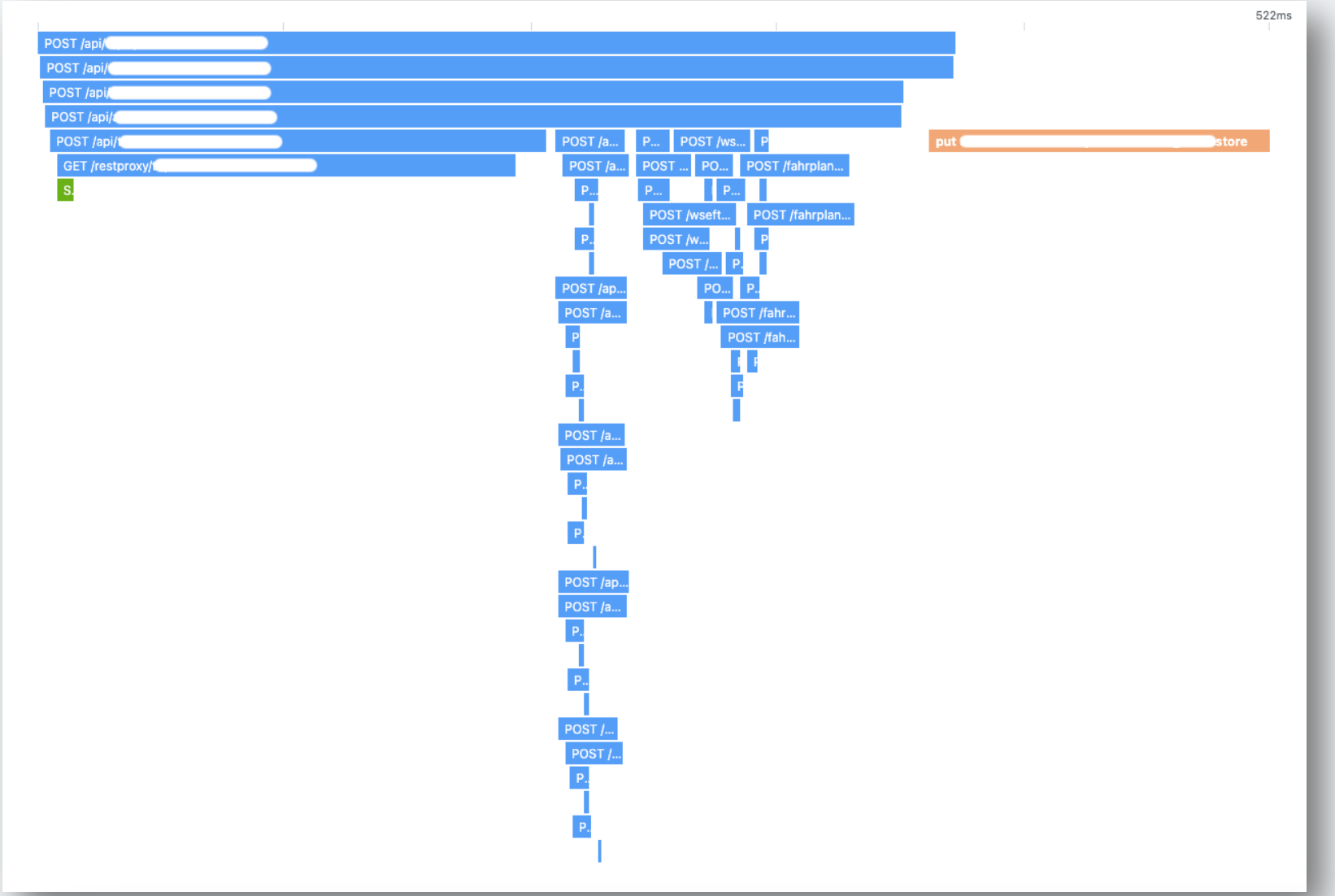
Herausforderungen: Große Antwortobjekte

- Pro Anfrage werden 5 bis max. 10 Verbindungen zurückgegeben
- Pro Verbindung werden mehrere Angebote zurückgegeben
- Jedes Angebot enthält Attribute für Darstellung, Buchung, Zahlung, Abrechnung, Materialisierung

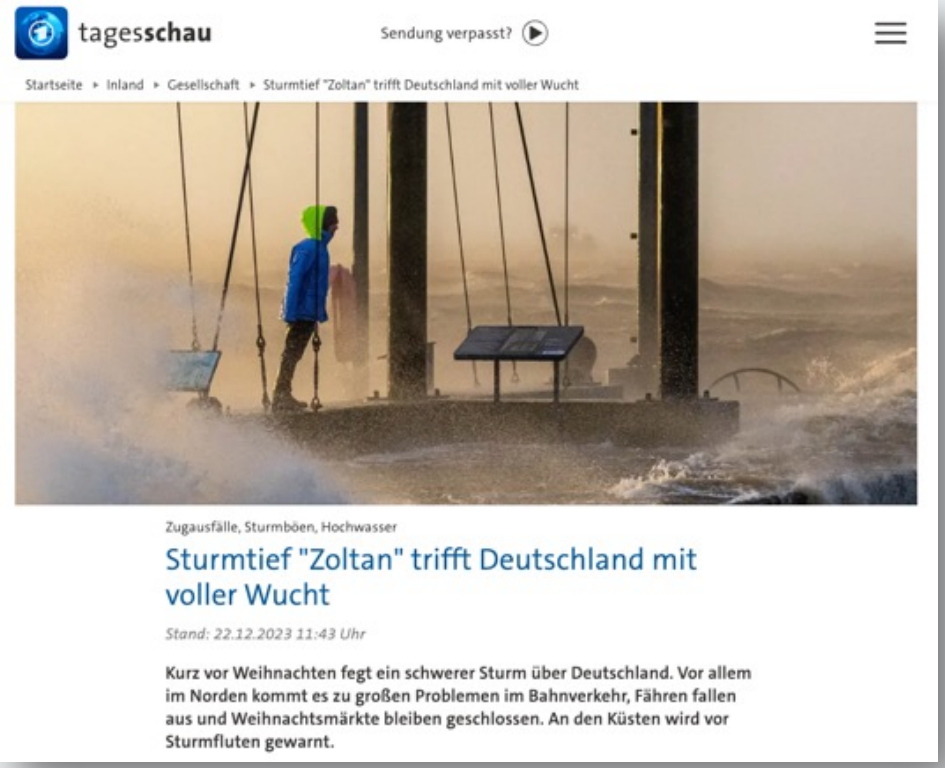


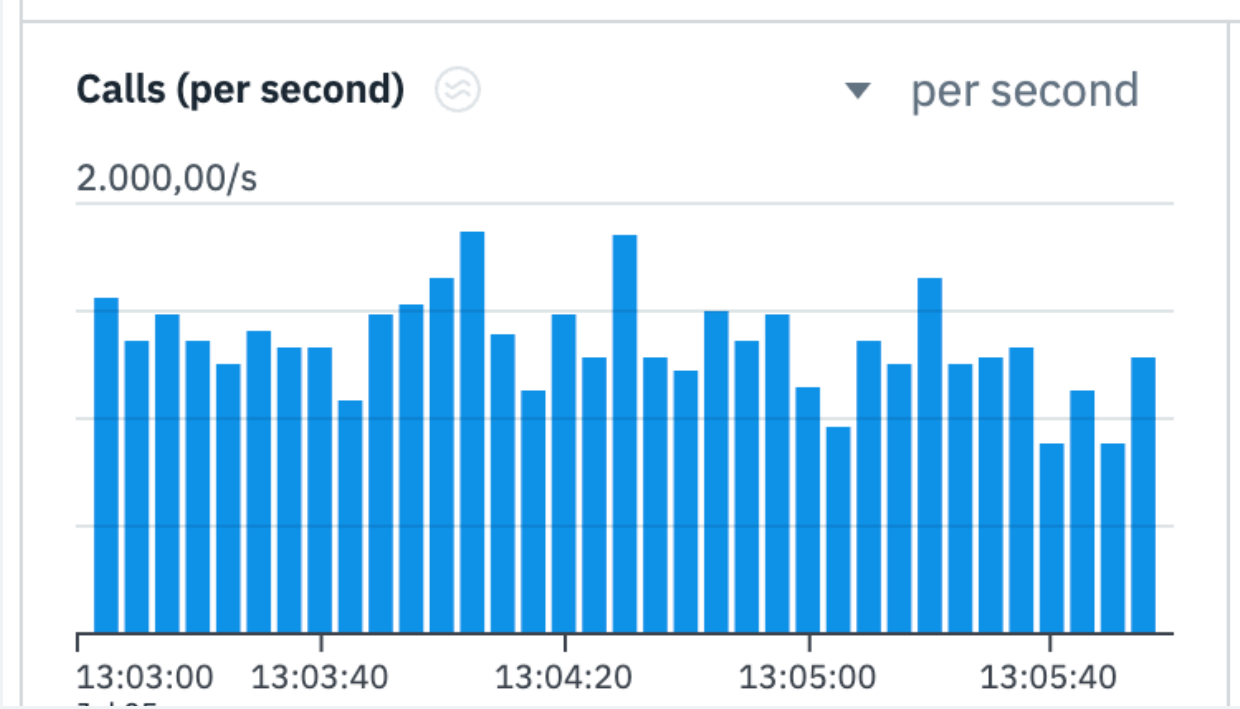


Herausforderungen: Hohe Anzahl von Netzwerk-Aufrufen



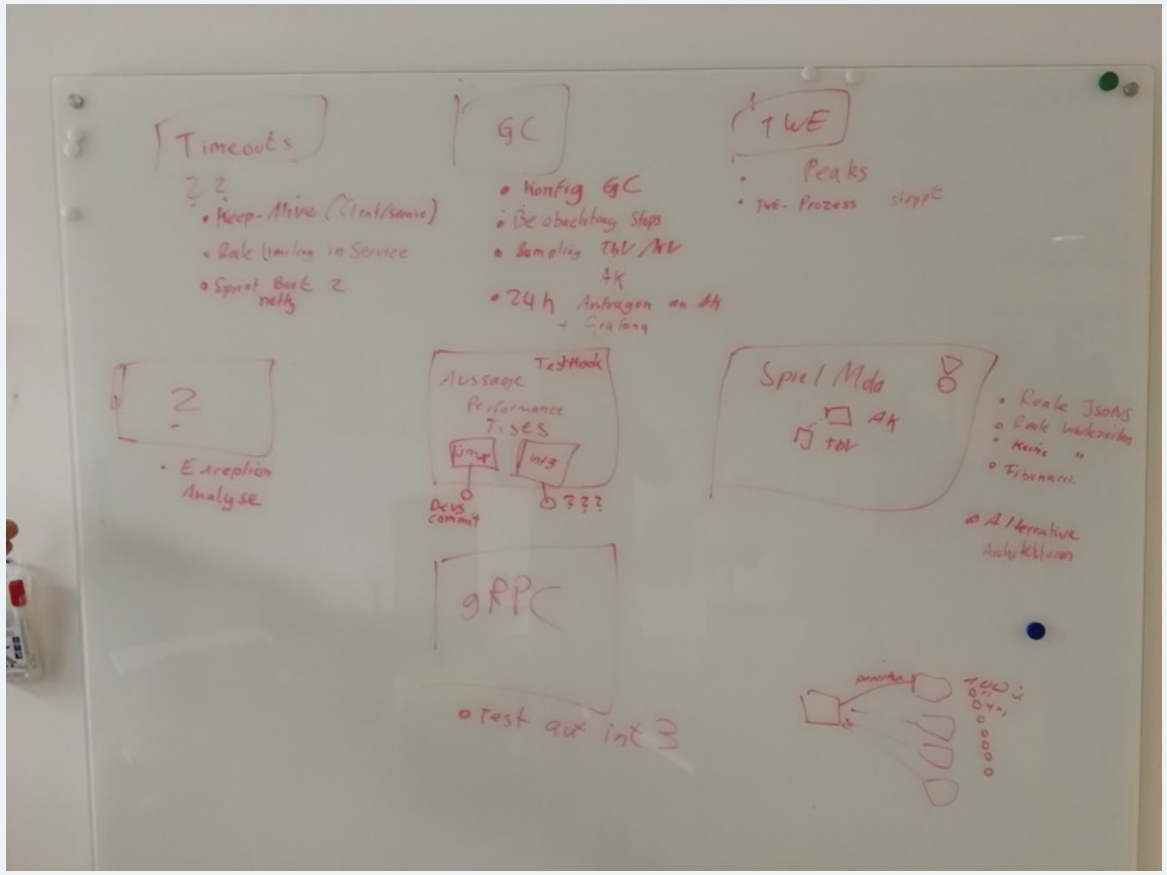
Herausforderungen: 2.000 Anfragen/Sekunde





https://www.tagesschau.de/inland/gesellschaft/sturmtief-zoltan-auswirkungen-102.html

Erste explizite Performance-Meetings



Vier apokalyptische Bottleneck-Reiter

Code

Netzwerk / Infrastruktur

Datenbank

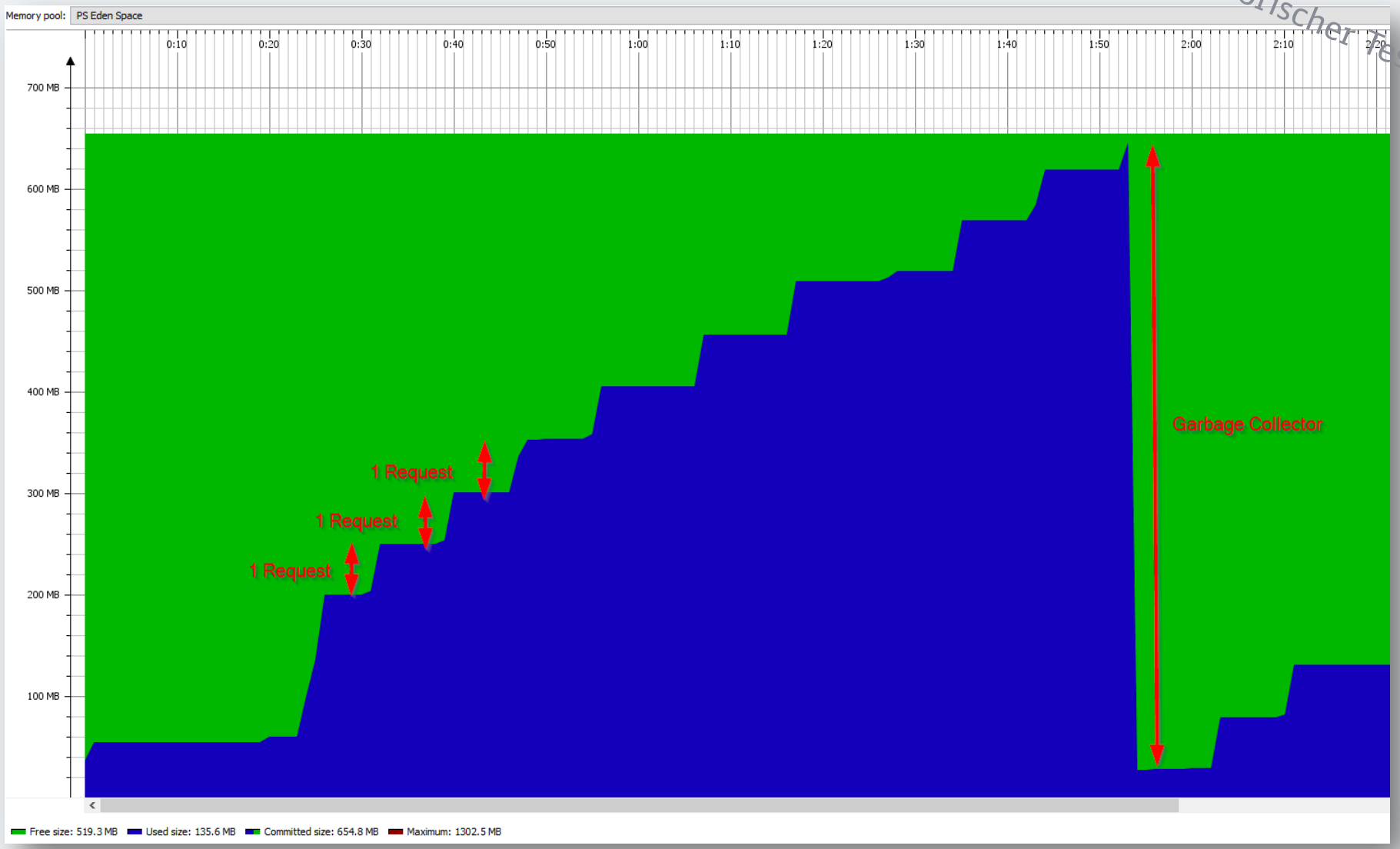
Architektur

(Beatus-von-Osma-Codex, 11. Jahrhundert)

Bottlenecks im Code: Logging

Bottleneck: Logging – Beispiel 1

Historischer Test: 2018



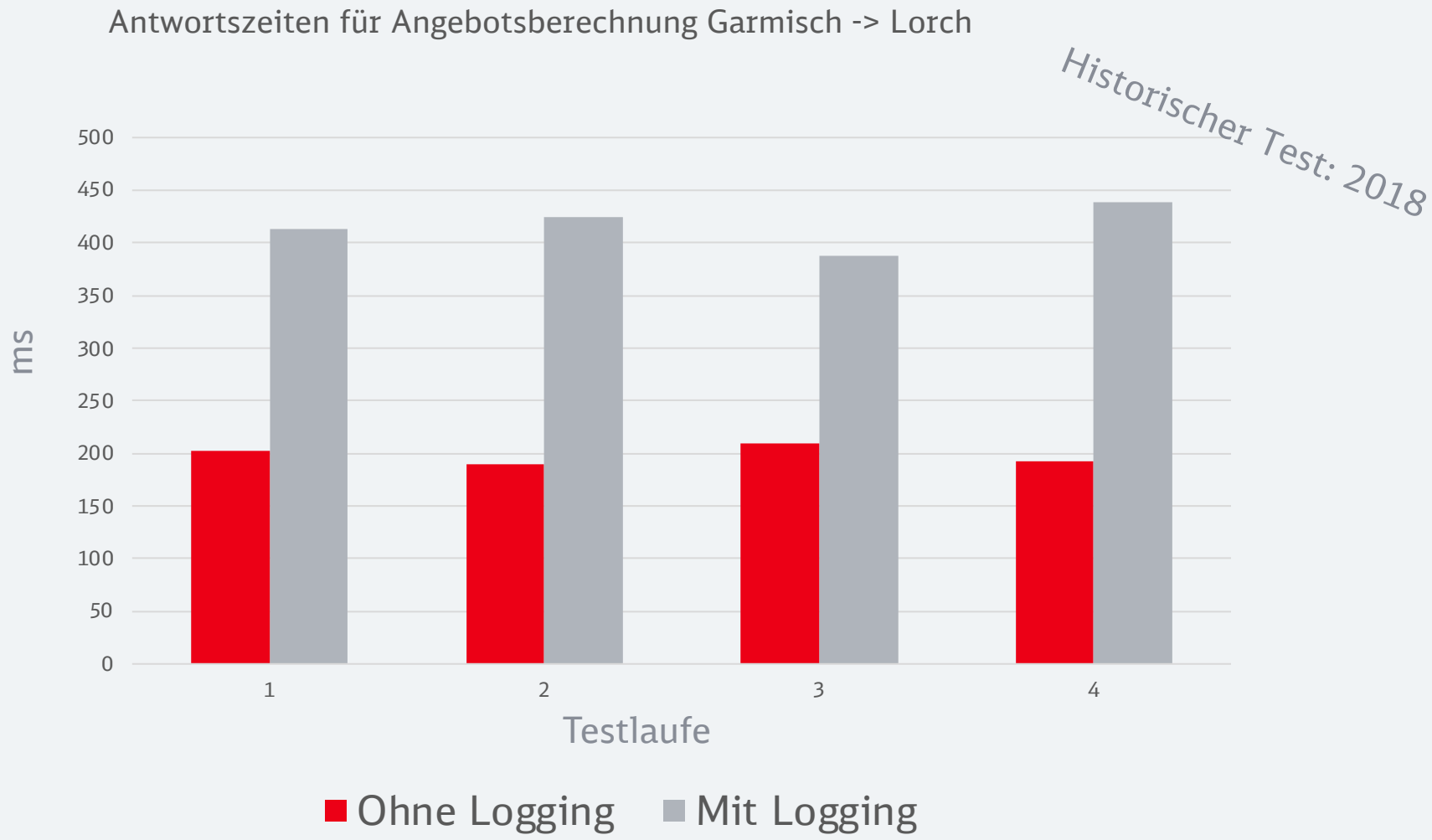
Logging – Probleme

Probleme beim Logging

- Jeder Berechnungsschritt wurde geloggt (z.B. auch der Rundungsschritt)
- Zu viele Strings werden erzeugt
- Aufgrund der Datenstrukturen sind die Strings sehr groß

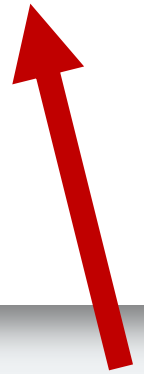
→ Zu viel Arbeit für die Garbage Collection

Bottleneck: Logging – Auswirkung von Entfernen jeder Logzeile



Mitigation Logging – Lazy Logging

```
log.debug( s: "Applied GesamtAngebotFilter Rules: {}", () ->
    filterRegeln.stream()
        .map(AngebotsRegel::getAngebotsRegelName)
        .collect(Collectors.toList())
);
```



Nur bei Loglevel DEBUG wird der String überhaupt erzeugt

Mitigation Logging – LoggingFilter

Filter als Log4J2-Plugin

```java
@Plugin(name = "DebugEnabledLoggerFilter", category = Node.CATEGORY, elementType = Filter.ELEMENT_TYPE, printObject = true)
@PerformanceSensitive("allocation")
@EqualsAndHashCode(callSuper = true)
public final class DebugEnabledLoggerFilter extends AbstractFilter {
```

```java
Result filter(final Level level, final String loggerName) {
    final String debugEnabledValue = injector.rawContextData().getValue(DEBUG_THREAD_CONTEXT_KEY);
    boolean shouldLog = Objects.equals(debugEnabledValue, b: "true")
        && level.isMoreSpecificThan(Level.DEBUG)
        && loggerName.startsWith(loggerNamePrefix);

    if (shouldLog) {
        return Result.ACCEPT;
    }

    return Result.NEUTRAL;
}
```

Konfiguration in der log4j2.xml

```xml
    <DebugEnabledLoggerFilter loggerNamePrefix="some.package"/>
    <Appenders>
```

Bottlenecks: Serialisierung

Jackson-Object-Mapper

```
public class ObjectMapper
extends ObjectCodec
implements Versioned, Serializable
```

ObjectMapper provides functionality for reading and writing JSON, either to and from basic POJOs (Plain Old Java Objects), or to and from a general-purpose JSON Tree Model (`JsonNode`), as well as related functionality for performing conversions. It is also highly customizable to work both with different styles of JSON content, and to support more advanced Object concepts such as polymorphism and Object identity. `ObjectMapper` also acts as a factory for more advanced `ObjectReader` and `ObjectWriter` classes. Mapper (and `ObjectReader`s, `ObjectWriter`s it constructs) will use instances of `JsonParser` and `JsonGenerator` for implementing actual reading/writing of JSON. Note that although most read and write methods are exposed through this class, some of the functionality is only exposed via `ObjectReader` and `ObjectWriter`: specifically, reading/writing of longer sequences of values is only available through `ObjectReader.readValues(InputStream)` and `ObjectWriter.writeValues(OutputStream)`.

Simplest usage is of form:

```
final ObjectMapper mapper = new ObjectMapper(); // can use static singleton, inject: just make sure to reuse!
MyValue value = new MyValue();
// ... and configure
File newState = new File("my-stuff.json");
mapper.writeValue(newState, value); // writes JSON serialization of MyValue instance
// or, read
MyValue older = mapper.readValue(new File("my-older-stuff.json"), MyValue.class);
```

https://www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-databind/2.15.3/com/fasterxml/jackson/databind/ObjectMapper.html

Bottleneck: „Wegwerf“-ObjectMapper – Beispiel 1

```java
@Mapper(
    disableSubMappingMethodsGeneration = true,
    unmappedTargetPolicy = ReportingPolicy.ERROR)
abstract class ZahlungsInfoMapper {

    ZahlungsInfo toApiZahlungsInfo(some.pack.age.ZahlungsInfo zahlungsInfo) {
        return new ObjectMapper().convertValue(zahlungsInfo, ZahlungsInfo.class);
    }
}
```

Benchmark mit Java Microbenchmark Harness (JMH)

```java
@Benchmark
@Measurement(iterations = 3)
@Fork(value = 1, warmups = 1)
@Warmup(iterations = 1)
public ZahlungsInfoDto newObjectMapper() {
    return new ObjectMapper().convertValue(zahlungsInfo, ZahlungsInfoDto.class);
}

@Benchmark
@Measurement(iterations = 3)
@Fork(value = 1, warmups = 1)
@Warmup(iterations = 1)
public ZahlungsInfoDto staticObjectMapper() {
    return objectMapper.convertValue(zahlungsInfo, ZahlungsInfoDto.class);
}
```

110.768 ops/s

6.441.972 ops/s

58x schneller

Gemessen mit Open JDK 21 / Jackson-Databind 2.16.0 / Macbook Pro M1
https://github.com/heikomaass/javaland_microbenchmarks

Jackson-Performance-Guide

Basics: Things You Should Do Anyway

There are some basic ground rules to follow to ensure that Jackson processes things efficiently (close to optimal level). These are things that you should "do anyway", even if you do not have actual performance problems: think of them as an interpretation of the "Boy Scout Rule" ("Always leave the campground cleaner than you found it"). Note that guidelines are shown in loosely decreasing order of importance.

1. Reuse heavy-weight objects: `ObjectMapper` (data-binding) and `JsonFactory` (streaming API)
    - To a lesser degree, you may also want to reuse `ObjectReader` and `ObjectWriter` instances -- this is just some icing on the cake, but they are fully thread-safe and reusable
2. Close things that need to be closed: `JsonParser`, `JsonGenerator`
    - This helps reuse underlying things such as symbol tables, reusable input/output buffers
    - Nothing to close for `ObjectMapper`
3. Use "unrefined" (least processed) forms of input: i.e., do not try decorating input sources and output targets:
    - Input: `byte[]` is best if you have it; `InputStream` second best; followed by `Reader` -- and in every case, do NOT try reading input into a String!
    - Output: `OutputStream` is best; `Writer` second best; and calling `writeValueAsString()` is the least efficient (why construct an intermediate String?)
    - Rationale: Jackson is very good at finding the most efficient (sometimes zero-copy) way to consume/produce JSON encoded data -- allow it do its magic
4. If you need to re-process, then replay and don't re-parse
    - Sometimes you need to process things in multiple phases; for example, you may need to parse part of JSON encoded data to plan out further processing or data-binding rules, and/or modify intermediate presentation for further processing

https://github.com/FasterXML/jackson-docs/wiki/Presentation:-Jackson-Performance

Bottleneck: Application Start-Up

Probleme mit Just-in-Time-Compilation

Erste Calls direkt nach dem Hochfahren eines Java-Prozess sind um Faktoren langsamer

	1	2	3	4	5	6	7	8	9
Ohne Warmup	486,538	40,134	35,678	26,441	25,358	25,551	23,725	23,435	22,375

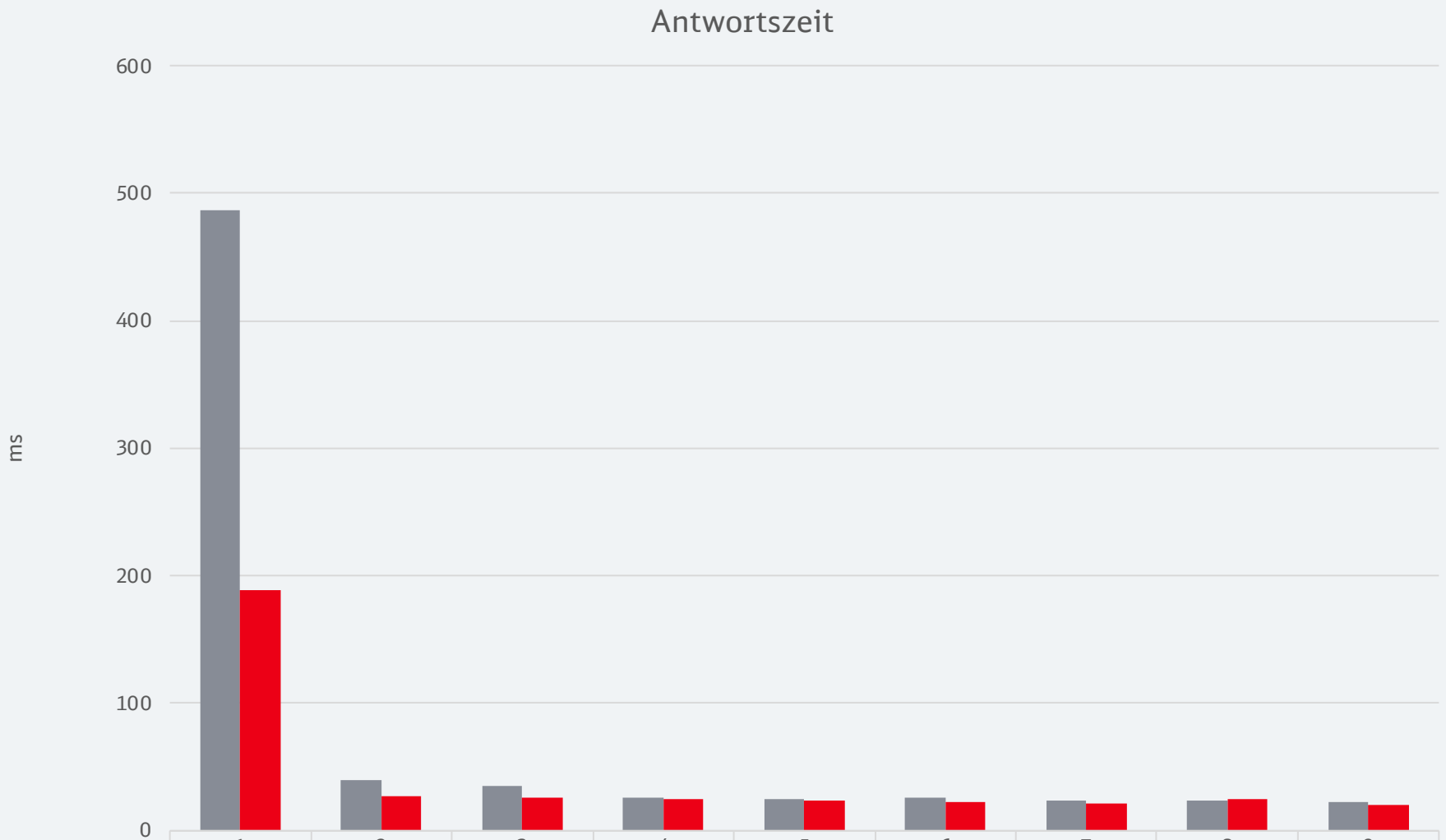
Mitigation: Warmup-Caller

- SpringBean: Im Post-Construct werden unterschiedliche Warm-Up-Szenarien ausführt

```
private void warmUpFahrplanbasiert() throws IOException {
    try (InputStream anfrageStream = anfrageResource.getInputStream()) {
        for (int i = 0; i < 10; i++) {
            FahrplanbasierteAnfrage anfrage = objectMapper.readValue(anfrageStream, FahrplanbasierteAnfrage.class);

            runWithWarmUpCorrelationId(() -> {
                Angebote angebote = fahrplanbasierteAnfrageController.getKatalogFahrplanAngebote(anfrage,  orgContextToken: null);
                logger.technisch().infoMessage("Angebote: " + angebote.getAngebote()).log();
            });
        }
    }
}
```

Verbesserte Antwortszeiten



	1	2	3	4	5	6	7	8	9
Ohne Warmup	486,538	40,134	35,678	26,441	25,358	25,551	23,725	23,435	22,375
Mit Warmup	188,506	27,373	26,245	24,971	23,517	22,95	21,946	25,004	19,834

Ausblick: Coordinated Restore at Checkpoint (CRaC)

- Offizielles OpenJDK Feature
- Von einer laufenden Anwendung wird ein „Snapshot“ erstellt
- Sockets/Files/Pools werden geschlossen und beim Starten der Snapshots wieder hergestellt

- Herausforderungen:
  - Erstellung des Snapshots mit Produktions-Konfiguration
  - Credentials stehen im Snapshot

Zusammenfassung Code-Bottlenecks

- Kenne die Auswirkungen der eingesetzten **Libraries** auf die Performance
- Finde einen Trade-off zwischen Komfort und Performance
- Berücksichtige den Just-In-Time Compiler-Effekt

Code

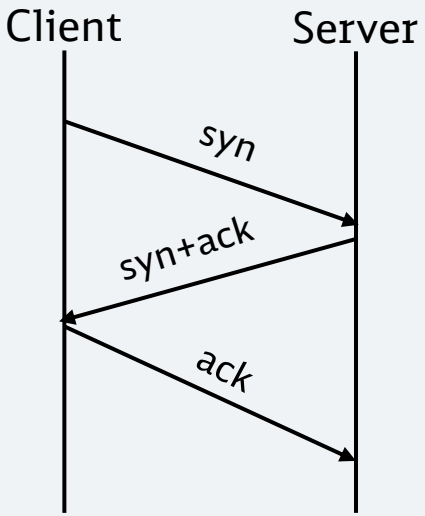
Netzwerk / Infrastruktur

Datenbank

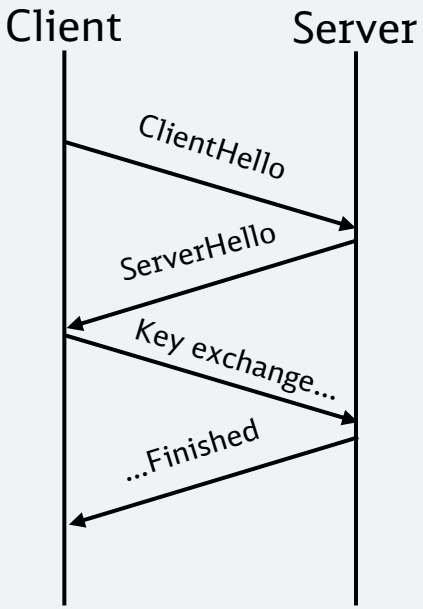
Architektur

Netzwerk / Infrastruktur

Overhead bei Connection-Aufbau



TCP-Handshake



TLS-Handshake

TCP Slow Start

Persistent Connections
(aka Keep-Alive)

Persistent-Connection (Keep-Alive) – Konfiguration erforderlich

- Standardmäßig nutzt das Spring den JDK-HttpClient **ohne** Persistent Connections / Connection-Pooling
- Konfiguration in Http-Clients notwendig. Beispiel Apache Commons HttpClient:

```java
@Bean
public CloseableHttpClient httpClient() {
    PoolingHttpClientConnectionManager connectionManager = new PoolingHttpClientConnectionManager();
    connectionManager.setMaxTotal(50); // default is 25
    connectionManager.setDefaultMaxPerRoute(50); // default is 5

    return HttpClients.custom()
            .setConnectionManager(connectionManager)
            .setKeepAliveStrategy((response, context) -> TimeValue.ofSeconds(50))
            .build();
}
```

Persistent-Connection (Keep-Alive) – Positive Auswirkung auf Latenzen

Historischer Test: 2019

15. Oktober 2019

Heiko Maaß (MDA) @heiko.maass 13:31

Vor dem Connection-Pooling:

```
11:17:20  > response time 50th percentile                208 (OK=208    KO=-     )
11:17:20  > response time 75th percentile                300 (OK=300    KO=-     )
11:17:20  > response time 95th percentile                550 (OK=550    KO=-     )
11:17:20  > response time 99th percentile                778 (OK=778    KO=-     )
11:17:20  > mean requests/sec                         99.174 (OK=99.174 KO=-     )
```

Nach dem Connection-Pooling:

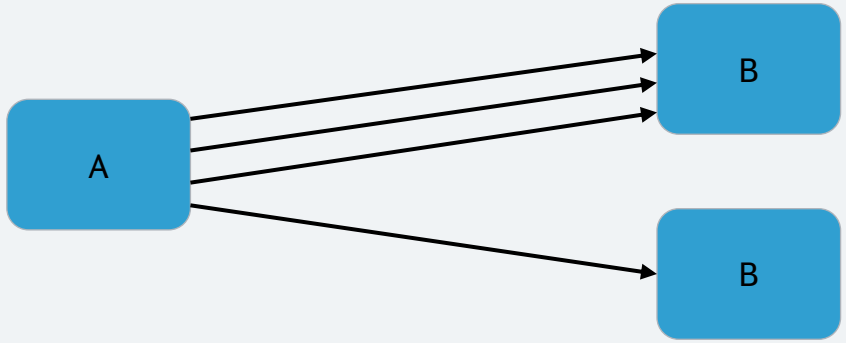
```
13:22:14  > response time 50th percentile                169 (OK=169    KO=-     )
13:22:14  > response time 75th percentile                193 (OK=194    KO=-     )
13:22:14  > response time 95th percentile                272 (OK=272    KO=-     )
13:22:14  > response time 99th percentile                359 (OK=359    KO=-     )
13:22:14  > mean requests/sec                         99.174 (OK=99.174 KO=-     )
```

Keep-Alive und  kubernetes

Persistent-Connection (Keep-Alive) – Load Balacing

Problem beim Load-Balancing

- Pods werden ungleichmäßig ausgelastet.
- Ältere Pods bekommen mehr Anfragen als neuere Pods „Sticky Connections“



Mitigation

- Setzen einer maximalen Lebensdauer von Connections (**maxLifeTime** in Netty)
- Server gibt bei Überlast den HTTP-Statuscode 429 zurück (Too Many Requests)
- Client schließt die Connection bei HTTP-Statuscode 429
- Langfristige Lösung: Service-Mesh

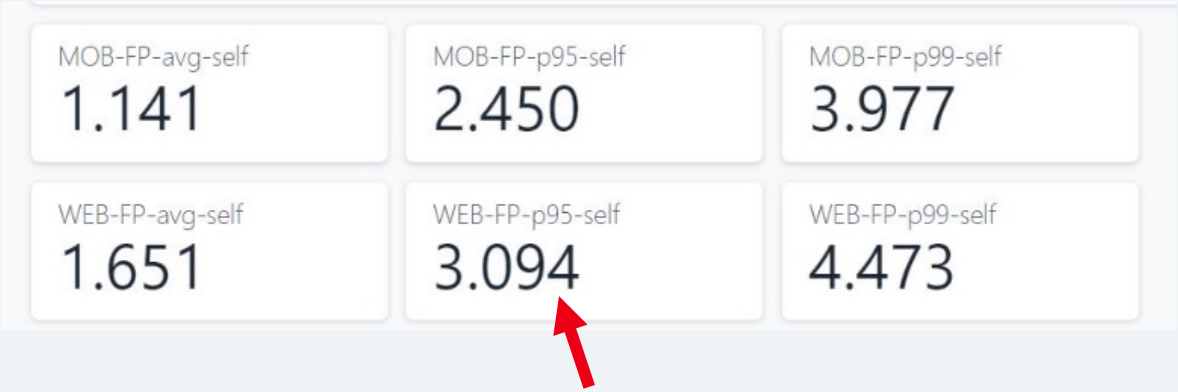
CPU-Throttling in kubernetes

Kubernetes CPU-Limits

- Definieren, ab wieviel CPU-Verbrauch der Pod "gedrosselt" wird.
- Die Drosselung hat negative Auswirkungen auf die Latenz und den Durchsatz, da der Pod während der Drosselung keine CPU-Zyklen erhält
- Problem: Drosselung findet oftmals zu **aggressiv** statt (selbst mit **aktuellem** Kernel)

Abschalten von CPU-Limits ?

Mit CPU-Limit



Ohne CPU-Limit

Abschalten von CPU-Limits – Nebeneffekte in Java

Nebeneffekt

- Runtime.getRuntime().availableProcessors() gibt Anzahl der CPUs des Hosts zurück
- Netty steuert mehrere Thread-Pools über diesen Wert
- JVM steuert Anzahl der GC-Threads über diesen Wert

```
reactor-http-epoll-48 id=113 state=RUNNABLE (running in native)
    at io.netty.channel.epoll.Native.epollWait(Native Method)
    at io.netty.channel.epoll.Native.epollWait(Native.java:209)
    at io.netty.channel.epoll.Native.epollWait(Native.java:202)
    at io.netty.channel.epoll.EpollEventLoop.epollWaitNoTimerChange(EpollEventLoop.java:316)
    at io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:373)
    at io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:997)
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.base@17.0.10/java.lang.Thread.run(Thread.java:840)

reactor-http-epoll-47 id=112 state=RUNNABLE (running in native)
    at io.netty.channel.epoll.Native.epollWait(Native Method)
    at io.netty.channel.epoll.Native.epollWait(Native.java:209)
    at io.netty.channel.epoll.Native.epollWait(Native.java:202)
    at io.netty.channel.epoll.EpollEventLoop.epollWaitNoTimerChange(EpollEventLoop.java:316)
    at io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:373)
    at io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:997)
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.base@17.0.10/java.lang.Thread.run(Thread.java:840)

reactor-http-epoll-46 id=111 state=RUNNABLE (running in native)
    at io.netty.channel.epoll.Native.epollWait(Native Method)
    at io.netty.channel.epoll.Native.epollWait(Native.java:209)
    at io.netty.channel.epoll.Native.epollWait(Native.java:202)
    at io.netty.channel.epoll.EpollEventLoop.epollWaitNoTimerChange(EpollEventLoop.java:316)
    at io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:373)
    at io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:997)
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.base@17.0.10/java.lang.Thread.run(Thread.java:840)
```

Abschalten von CPU-Limits – Nebeneffekte in Java

Mitigation

- Alternative 1:
  - Explizite Konfiguration der Threadpool-Größen
  - Explizite Konfiguration der Anzahl der Garbage-Collection-Threads

```
-XX:ParallelGCThreads=4
-Dreactor.schedulers.defaultPoolSize=4
-Dreactor.netty.ioWorkerCount=4
```

- Alternative 2:
  - Setzen von -XX:ActiveProcessorCount=4

Abschalten von CPU-Limits – Risiken

Risiko

- Ein Pod frisst dauerhaft mehr CPU als requested -> Beeinträchtigt alle Pods auf dem Knoten.

Mitigation

- Abschalten nur für latenz-kritische Services
- Abschalten von CPU-Limits nur mit aktiviertem HorizontalPodAutoscaling

Mögliche Alternativen

- Verproben von cgroups v2 (https://kubernetes.io/docs/concepts/architecture/cgroups/#check-cgroup-version)
- CPU Management Policies on the Node (https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/)

Zusammenfassung Infrastruktur-Bottlenecks

- Reduziere Netzwerk-Overhead
- Eliminiere/Reduziere CPU-Throttling
- Berücksichtige Seiteneffekte
  - mit Keep-Alive
  - ohne CPU-Limits

Code

Netzwerk / Infrastruktur

Datenbank

Architektur


Architektur-Änderungen

Zu viele parallele Threads in einem Service

Probleme bei der Konzentration von Services

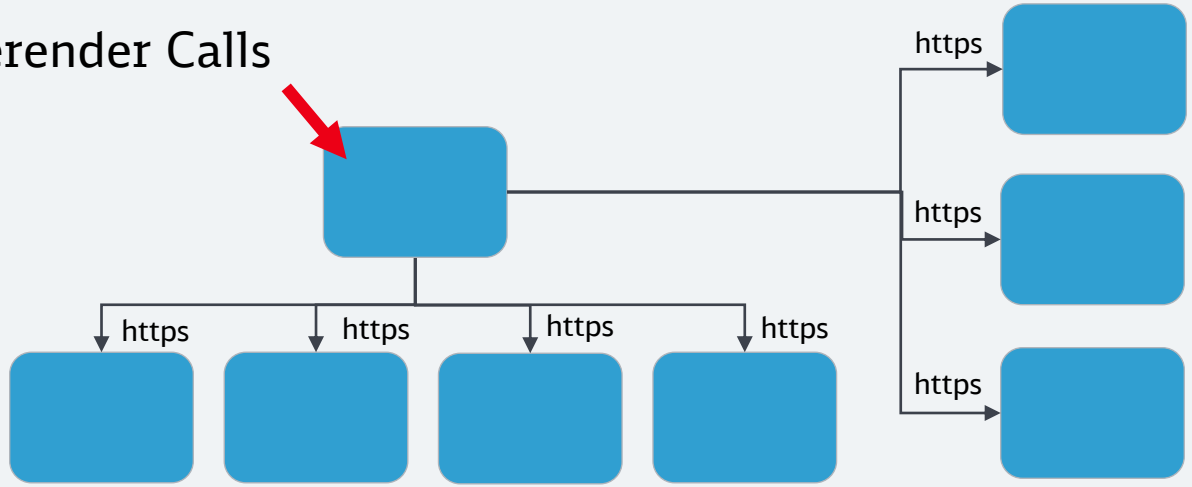
- Pro ausgehendem Call wird ein Thread blockiert
- Threads kosten Betriebssystem-Ressourcen

Umgesetzte Lösung im Jahr 2020

- Umstellung von Spring MVC auf Spring **Webflux**
- Reaktives Programmiermodell anstelle blockierender Calls
- Größeres Refactoring

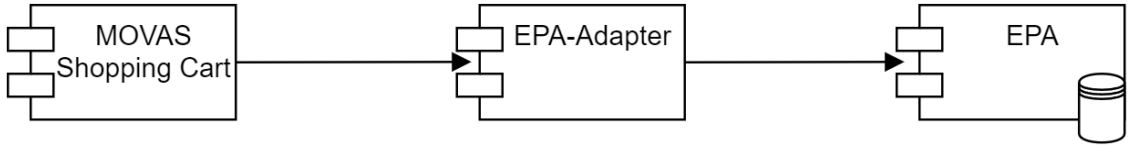
Lösungsvorschlag im Jahr 2024

- **Keine** Migration auf Webflux
- Nutzung Virtual Threads-Feature des JDK 21



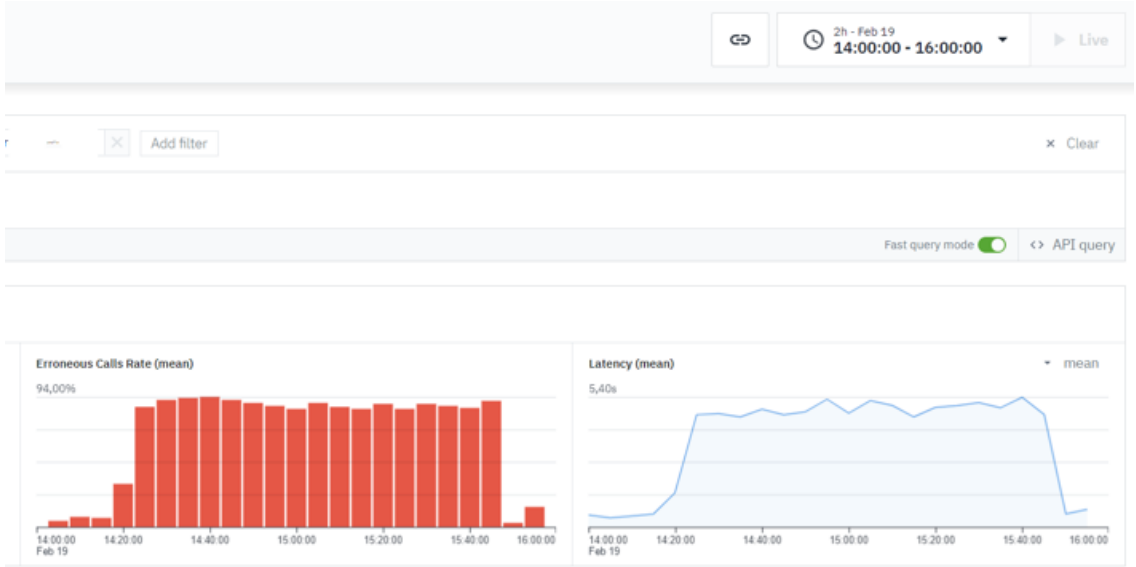
Integration von Dritt- und Legacy-Systemen

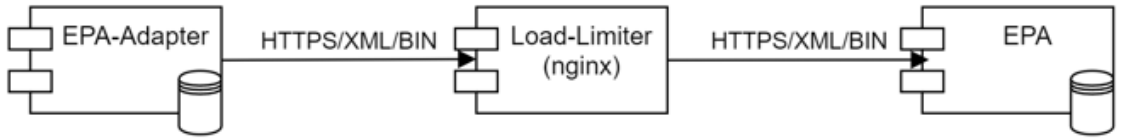
- Exkurs: 40 Jahre altes Sitzplatzreservierungssystem EPA
- teilweise sehr große technologische Gaps
- NFA von großer Bedeutung
- Middleware + Kompatibilität beachten



Last & Überlast

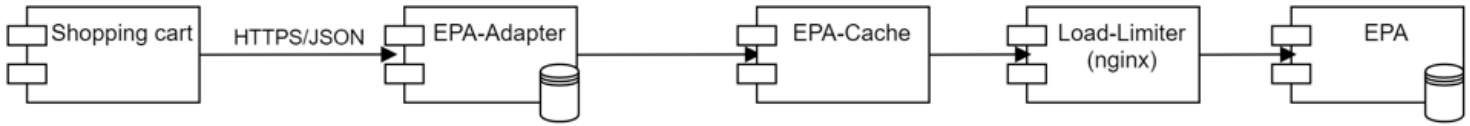
- EPA verfügt über keine Schutzmechanismen
- Kein „graceful degradation“ bei Überlast
- ➡️ Service Mesh sidecar Container?
- ➡️ eigene Komponente zur Lastbegrenzung erforderlich
- in unserem Fall: nginx
- „gute“ Limits bestimmen: Lasttests (in PRD)

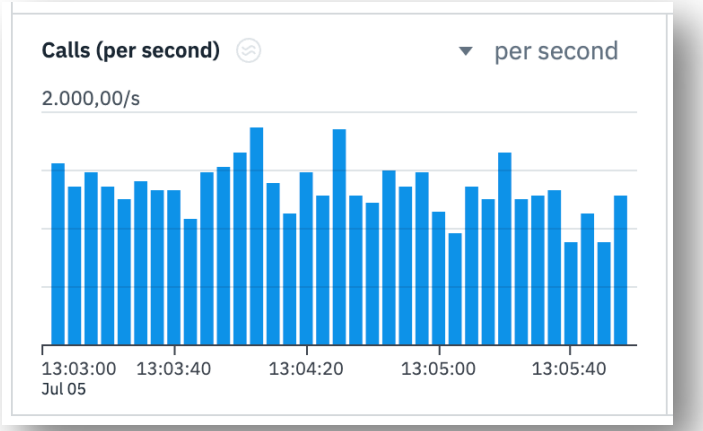


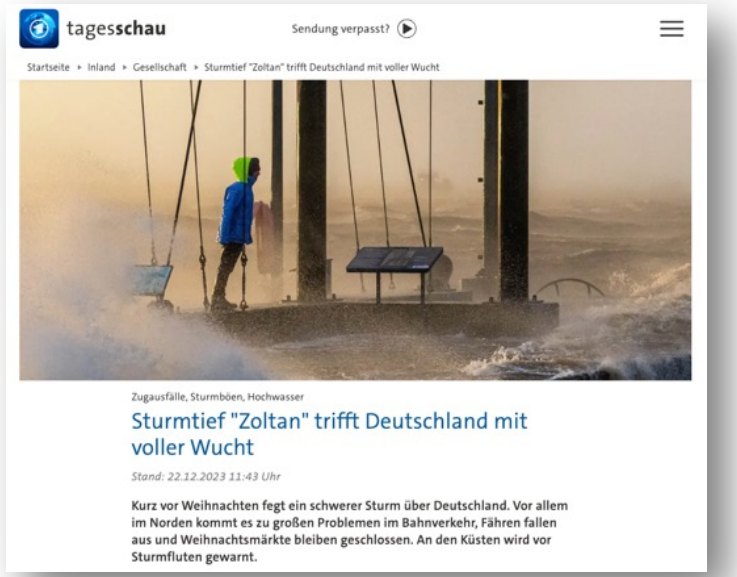


Spitzenlast

- Peaks von über 2.000 req/s
- EPA-Grenzen liegen um einiges niedriger
- ➡️ überwiegender Teil der Requests laufen in HTTP 429







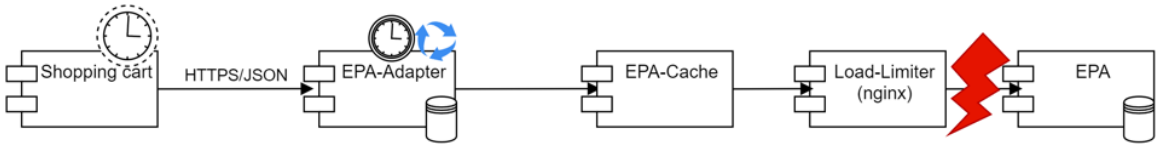
TLS Connection Keepalive

- N.b.: Connection Keepalive standardmäßig **deaktiviert** in Nginx
- ➡️ “**Avoiding the Top 10 NGINX Configuration Mistakes” (#3)**
- SSL session cache & SSL session timeout spielen ebenfalls wichtige Rolle bei großen Mengengerüsten

```
upstream ${BACKEND_URL} {
    server ${BACKEND_URL}:${BACKEND_PORT} max_conns=${KEEPALIVE_UPSTREAM_CONNECTIONS};
    keepalive ${KEEPALIVE_UPSTREAM_CONNECTIONS};
    keepalive_requests ${KEEPALIVE_UPSTREAM_REQUESTS};
    keepalive_timeout ${KEEPALIVE_UPSTREAM_TIMEOUT};
}
```

Timeouts & verteilte Transaktionen

- Schreiboperationen als verteilte Transaktionen
- Saga Pattern?
- 2PC / XA?
- Eventual consistency?



Zusammenfassung Architektur-Bottlenecks

- (Legacy)-Integration i.d.R. doch aufwändiger als auf den ersten Blick
- Insb. nicht-funktionale Anforderungen und technologische Herausforderungen berücksichtigen
- Oft sind „kreative“ Lösungen gefragt
- Dringende Empfehlung: explizites Management von Wissen und Software Lifecycle

Epilog

- Man kann die Bottleneck-Reiter nicht dauerhaft ausschalten, sondern nur zurückdrängen
- Mit jeder neuen Anforderung können sie wieder zurückkommen (Wiedergänger)
- Performance-Optimierung als dauerhaften Prozess
- Technische Voraussetzungen notwendig:
  - Möglichst früh automatisierte Lasttests
  - Einblick in die Infrastruktur-Metriken (CPU-Throttling, Threads, Profiling)
  - Tracing aller Calls
- Automatische Überprüfungen auf bekannte Bottlenecks

Vielen Dank !

heiko.maass@deutschebahn.com
lukas.pradel@deutschebahn.com

https://github.com/heikomaass/javaland_microbenchmarks

https://github.com/heikomaass/javaland_mesobenchmarks