Boris Fresow, Markus Günther

# Advanced Spring Workshop

Adesso eduCamp 2023

11 – 13 May 2023 / Mastichari, Kos

# Day 1
# Advanced Spring
## Spring Cloud

# Advanced Spring

## Organisation and Workshop Agenda

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### Who are we?

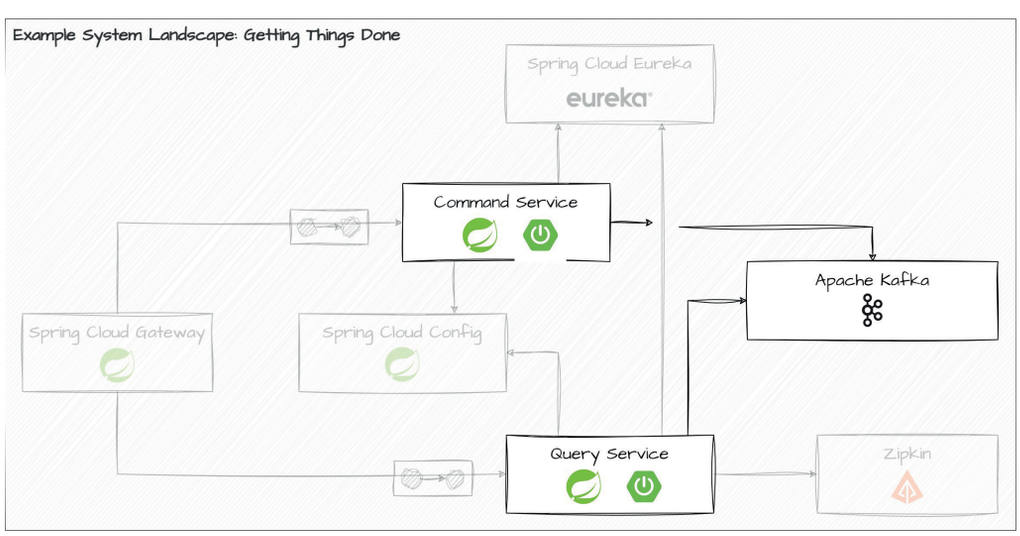
**Boris Fresow** is a freelance IT consultant and trainer. He mainly deals with distributed systems, messaging solutions, and training on these topics.

**Markus Günther** is a freelance software developer, architect, and trainer. He supports his clients in implementing robust, scalable systems and also offers seminars & workshops on messaging solutions and modern software methodology.

# Advanced Spring – Organisation and Workshop Agenda

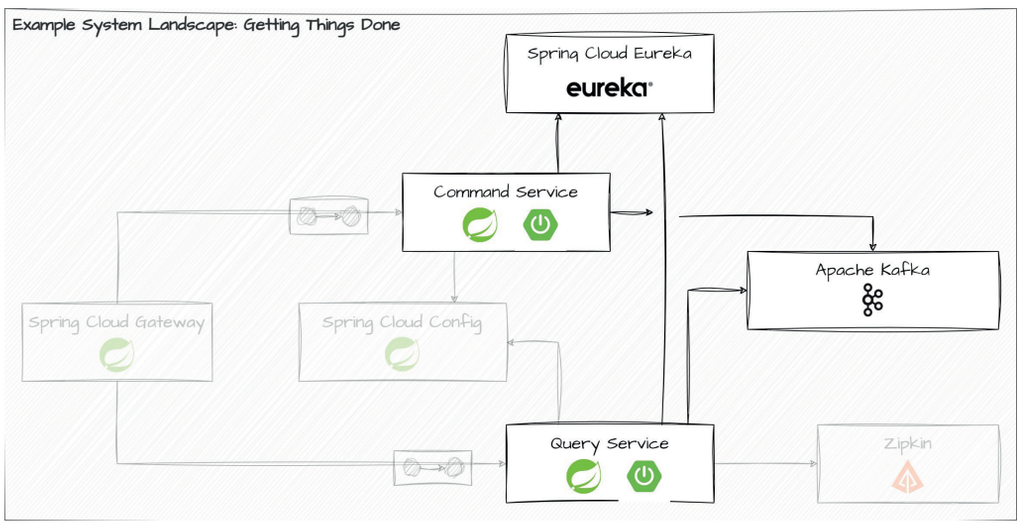
| | Day 1 | Day 2 | Day 3 |
|---|---|---|---|
| 09:00<br>09:15 | Organisation, Workshop Agenda, Goals | Recap Day 1 | Recap Day 2 |
| 09:30<br>09:45<br>10:00<br>10:15 | Introduction to Spring Cloud | Principles of Security in Distributed Systems | Spring Boot Migration Guide |
| 10:30<br>10:45 | Coffee Break | Coffee Break | Coffee Break |
| 11:00<br>11:15<br>11:30<br>11:45<br>12:00<br>12:15 | Service Discovery with Eureka | Spring Security | Testing with Spring - Best Practices |
| 12:30<br>12:45<br>13:00<br>13:15 | Lunch Break | Lunch Break | Lunch Break |
| 13:30<br>13:45<br>14:00<br>14:15<br>14:30<br>14:45 | Spring Cloud Gateway | Distributed Tracing with Spring | Monitoring Business-Related Metrics |
| 15:00<br>15:15 | Coffee Break | Coffee Break | Coffee Break |
| 15:30<br>15:45<br>16:00 | Spring Cloud Config | Secure Communication | Case Study & Wrap-Up |
| 16:15<br>16:30 | Resilience4j | Overflow | |
| 16:45<br>17:00 | Discussion / Q & A | Discussion / Q & A | |

## We'll be working on a simple todo application, ...



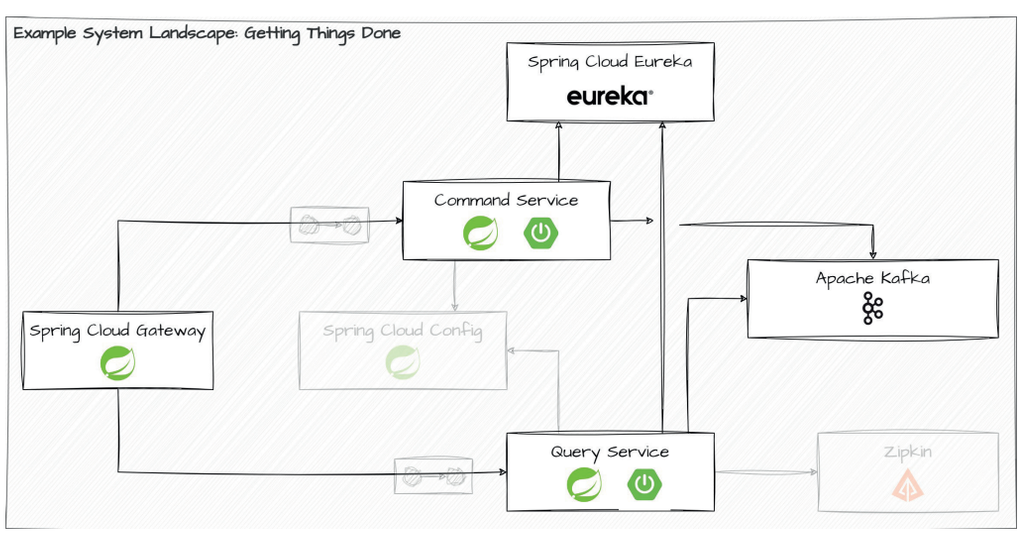
# Advanced Spring – Organisation and Workshop Agenda

## ... and introduce Spring Cloud components along the way.

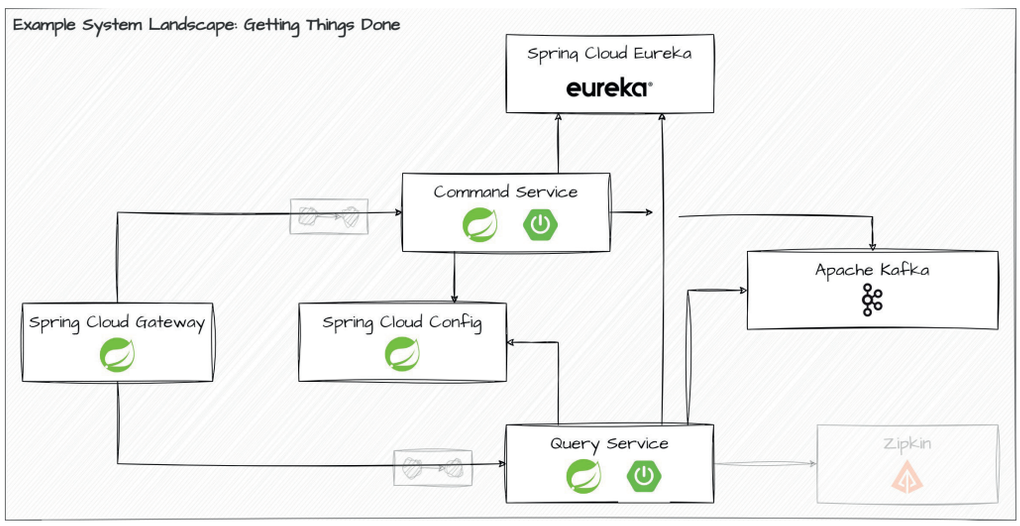





## ... and introduce Spring Cloud components along the way.

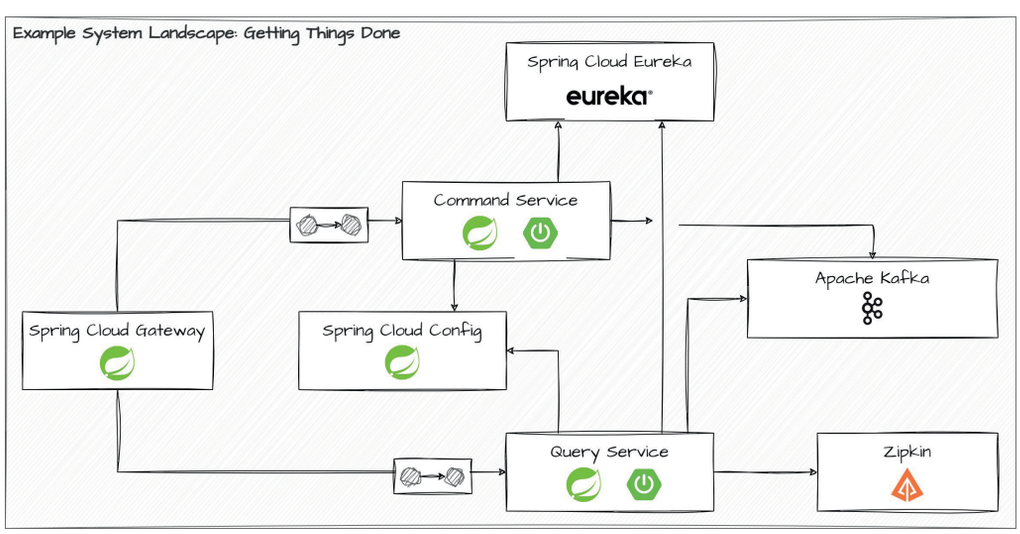


## ... and introduce Spring Cloud components along the way.






## ... and introduce Spring Cloud components along the way.



# Advanced Spring – Organisation and Workshop Agenda

## What we'll provide

- Material in terms of slides and accompanying notes
- Assignments for certain topics
  - Isolated GitHub repositories
  - `README.md`: General information
  - `ASSIGNMENT.md`: Tasks
  - `HINTS.md`: In case you're stuck
- Project files and / or Docker-based local environment



## What you'll need

- Working installation of
  - JDK 17
  - Docker and Docker Compose
- Preferably IntelliJ IDEA
  - doesn't matter if Community or Ultimate Edition

# Advanced Spring – Organisation and Workshop Agenda

## Goals

- Get some familiarity with Spring Cloud
- Know a thing or two about security
- Best Practices on testing Spring components

# Advanced Spring

## Introduction to Spring Cloud

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

## Cloud computing continues to grow and evolve.

- 94% of enterprises in the US use cloud services
- 67% of enterprise infrastructure is cloud-based
- 92% of businesses employ a multi-cloud strategy
- Cloud computing market size (global)
  - 2022: $480.04 billion
  - 2026: $947.30 billion

Source: 25 Amazing Cloud Adoption Statistics (2023): Cloud Migration, Computing, And More



- 25 Amazing Cloud Adoption Statistics (2023): Cloud Migration, Computing, And More is available at https://www.zippia.com/advice/cloud-adoption-statistics/

**The shift toward cloud computing comes with *organizational* challenges.**

- On-prem deployments play a lesser role
- No dedicated operations teams
- Shift toward a dev/ops culture
- Experience with cloud technologies is essential



Transitioning to cloud computing introduces organizational challenges for businesses. These difficulties can involve adapting to new technologies, processes and roles, as well as addressing security, cost, and compliance concerns during the shift.

**A shift toward cloud computing comes with *technical* challenges.**

- Automate node setup and configuration
- Up-to-date and consistent configuration
- Prevent cascading failures
- Monitoring the state of a system of services
- Collecting log files and correlating log events
- Complexity of distributed systems



(1) Adding new service instances required the manual configuration of load balancers and manually setting up new nodes. This is error-prone and repetitive and we'd like to automate these tasks in a highly dynamic environment.

(2) Keeping the configuration of all instances consistent and up-to-date involved manual and repetitive work. This is also a task that we'd have to automate in a cloud-based setup.

(3) Synchronous communication can lead to cascading failures. This means that clients of a crashing component could also crash after a while. This is known as chain of failures.

(4) Monitoring the state of the platform in terms of technical parameters, such as latency and hardware usage, is more complicated than monitoring a single instance of a monolithic application.

(5) Collecting log files from distributed services and correlating relevant log events is difficult.

(6) A distributed system is inherently more complex than a single server instance running on a dedicated machine.

**Don't fall for misconceptions when designing distributed systems.**

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) The network is reliable is the mistaken belief that communication in a distributed system will always occur without failures, overlooking potential issues such as network outages, packet loss, or hardware problems that can impact the system's reliability.

(2) The latency is zero is the erroneous assumption that data transmission within a distributed system occurs instantaneously, disregarding the real-world delays caused by factors such as network congestion, propagation time, and processing overhead.

(3) The bandwidth is infinite is the false belief that a distributed system's network has unlimited capacity to transmit data, neglecting the actual constraints imposed by factors like hardware limitations, network infrastructure, and competing traffic.

(4) The network is secure is the incorrect assumption that a distributed system's network is inherently protected from potential threats, overlooking the need to address vulnerabilities, implement security measures, and guard against malicious actors.

(5) The topology doesn't change is the mistaken belief that the structure of a distributed system's network remains static over time, ignoring the reality that nodes may be added or removed, and connections can change due to hardware failures, updates, or other factors.

(6) There is one administrator is the false assumption that a distributed system operates under centralized control, neglecting the fact that different components may be managed by multiple administrators or organizations, potentially leading to coordination challenges and conflicting policies.

(7) Transport cost is zero is the erroneous belief that communication within a distributed system comes without any overhead or expense, disregarding the real-world costs associated with bandwidth usage, energy consumption, and the resources required to establish and maintain network connections.

(8) The network is homogeneous is the incorrect assumption that all components within a distributed system use consistent protocols, standards, and technologies, overlooking the diversity of software, hardware, and network configurations that may exist and require interoperability solutions.

**Spring Cloud provides out-of-the-box solutions for common cloud-based challenges.**

- Inception: 2014
- Extends power and simplicity of Spring Boot to the cloud
- Inspired by Netflix OSS projects
- Grown significantly over the past years
- Integrations with Kubernetes, Consul, Istio, ...

## Spring Cloud provides out-of-the-box solutions for common cloud-based challenges. (cont.)

Spring Cloud is a popular choice for building

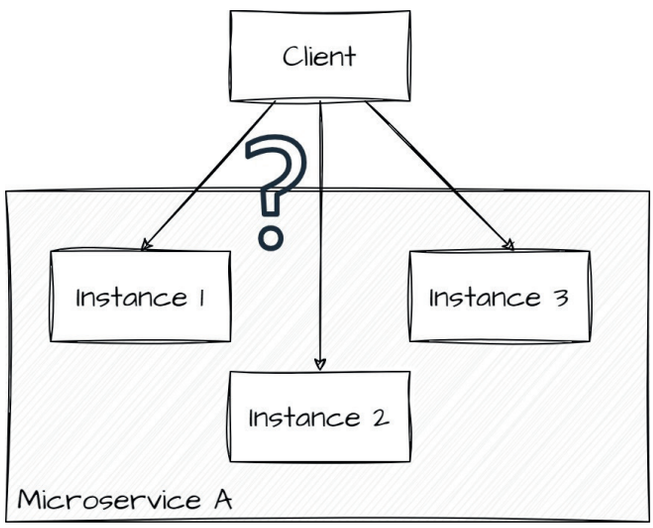
**resilient and scalable**

cloud-based applications in the Java ecosystem.



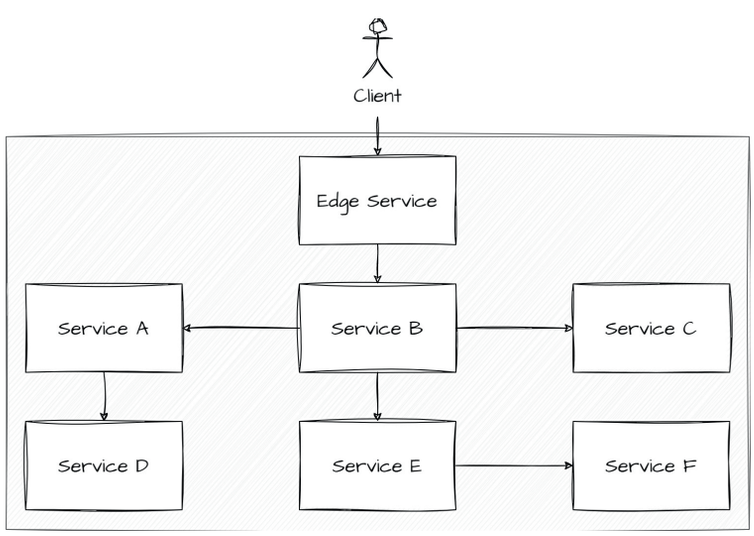
## Design Patterns and Spring Cloud

## A service discovery helps to look up services and their instances.

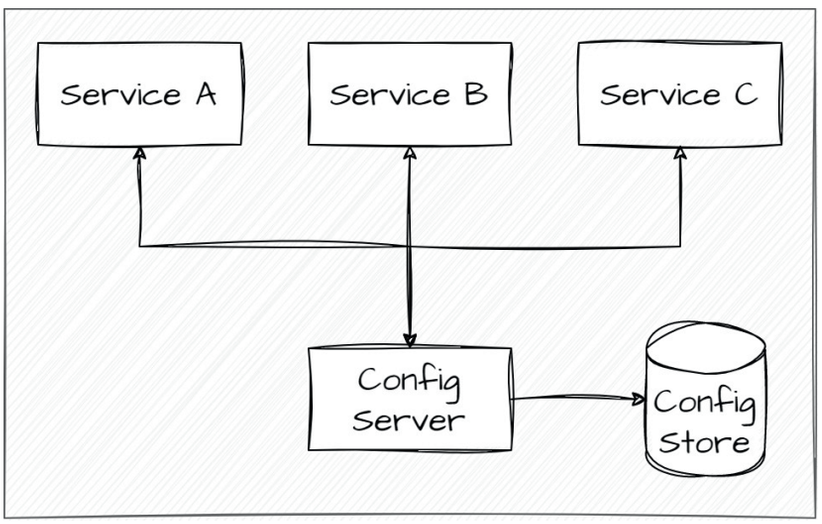





## An edge service hides internal services and secures those that are exposed.

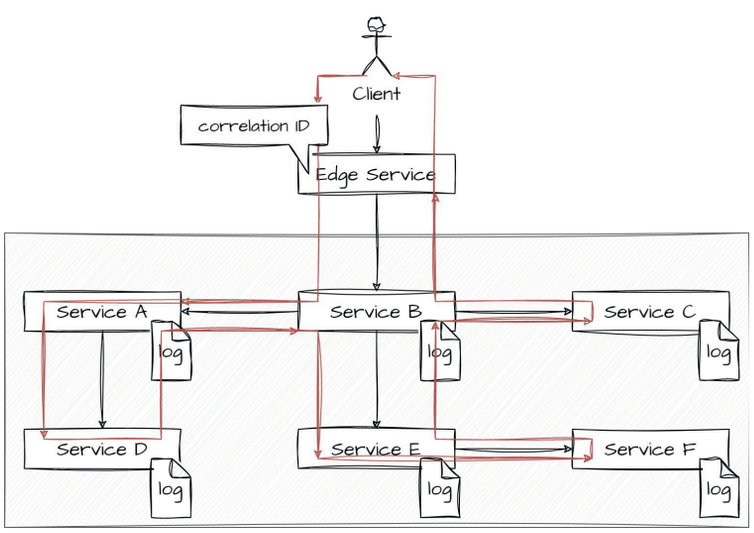


## A config server simplifies centralized configuration management and promotes consistency.

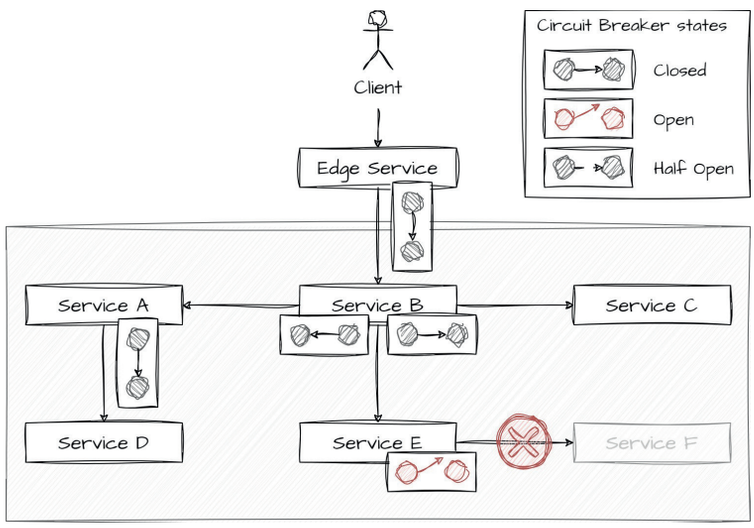





## Distributed tracing enables us to track requests and messages between services.



## A circuit breaker prevents from a chain of failure.



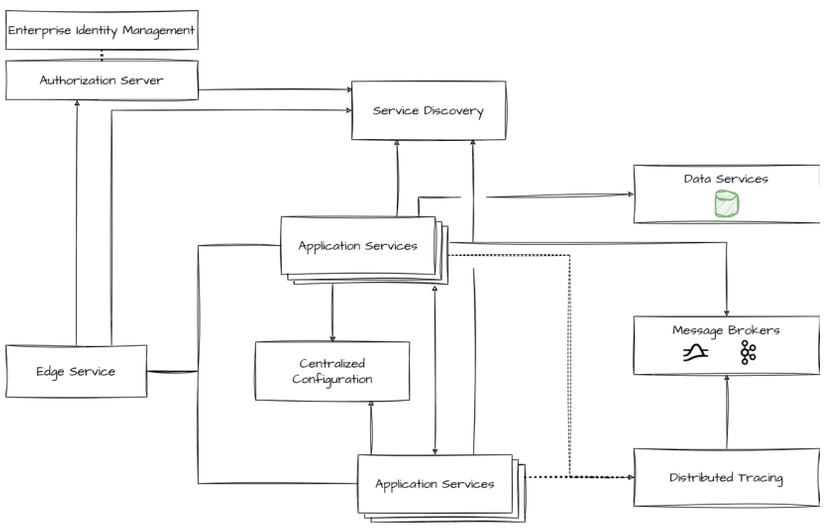



## But wait, there's more!

- Centralized Log Analysis
- Centralized Monitoring and Alarms
- More Resilience Patterns
- ...

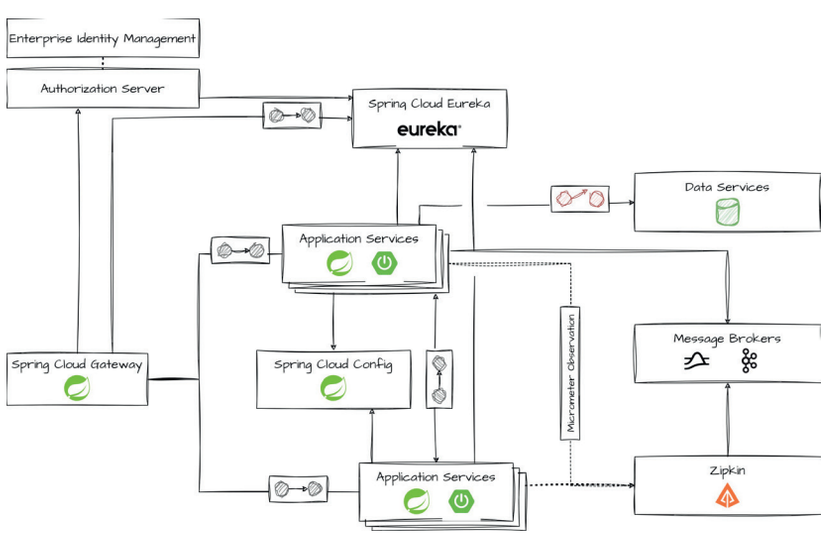
**we wont't discuss these in this workshop**

## Combining these design patterns addresses the challenges with Cloud-based services.

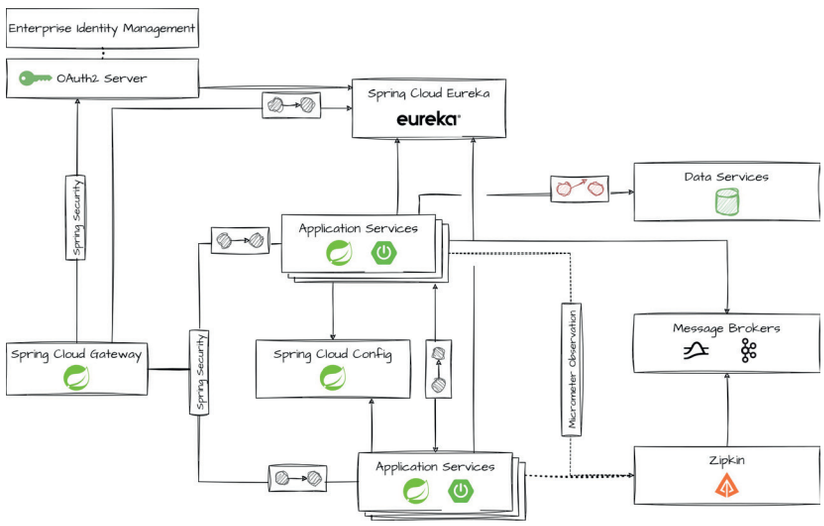





## Spring Cloud offers battle-proven software components to implement these patterns.



## Spring Security addresses security considerations in a highly-distributed environment.



# Advanced Spring

## Service Discovery with Eureka

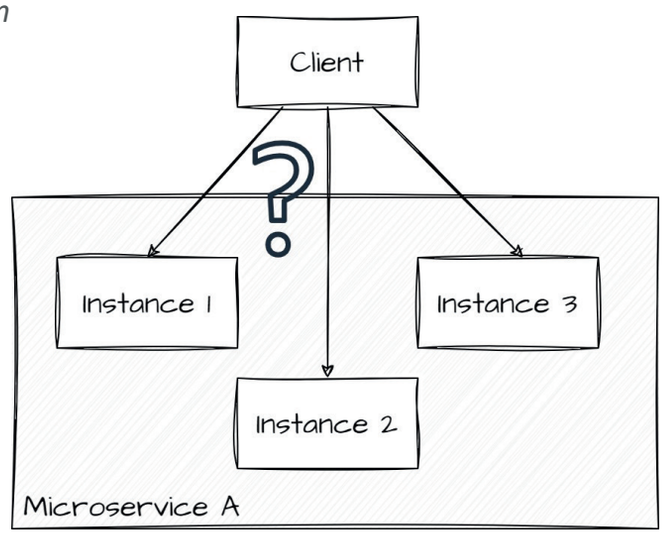
**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

## Why is service discovery important in Cloud-based architectures?

- Service instances suffer from *instance churn*
- IP addresses are dynamically allocated



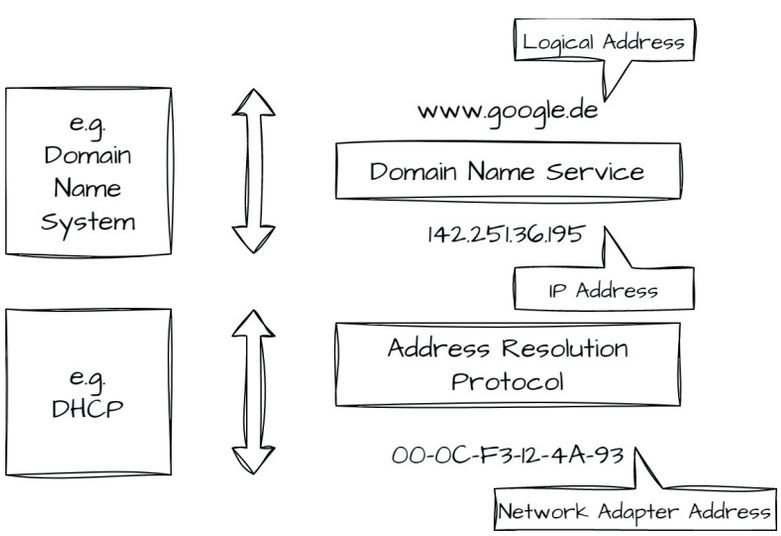

In a production-grade cloud setup, service instances have some degree of churn. They come and go dynamically, hence they are assigned dynamically allocated IP addresses. This makes it difficult for a client to submit a request to a service that, for example, exposes a HTTP API.

## Addressing networked entities includes names, addresses and routes.

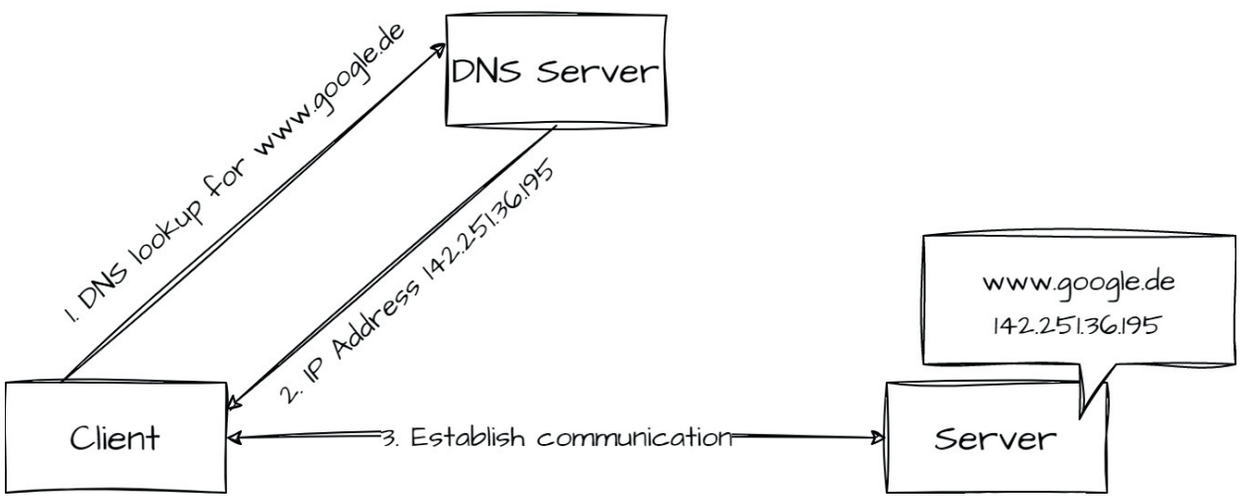
*The **NAME** of a resource indicates **WHAT** we seek, an **ADDRESS** indicates **WHERE** it is, and a **ROUTE** tells us **HOW TO GET THERE**.*



## Addressing occurs at several levels of abstraction.



## A short reminder on how the Domain Name System works.






## Why not use (round-robin) DNS for service discovery?

1. DNS is not suitable for volatility (instance churn).
2. DNS clients don't do round-robin per request.
3. Security

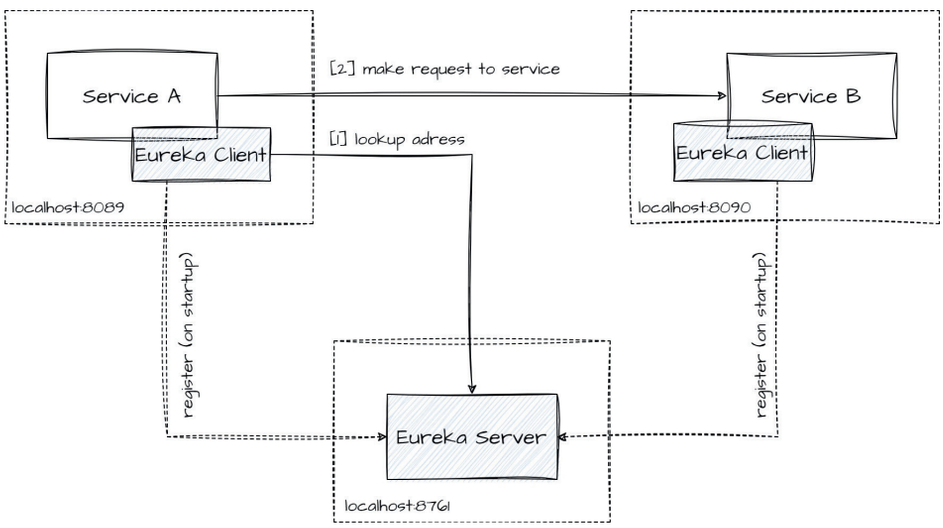
**Service discovery presents us with a couple of tough challenges.**

- New instances can start up at any point in time.
- Existing instances can stop responding.
- Failing instances might be okay after a while.
- Instances may have a high startup delay.
- Network partitions disrupt connectivity.



## Service Discovery with Netflix Eureka

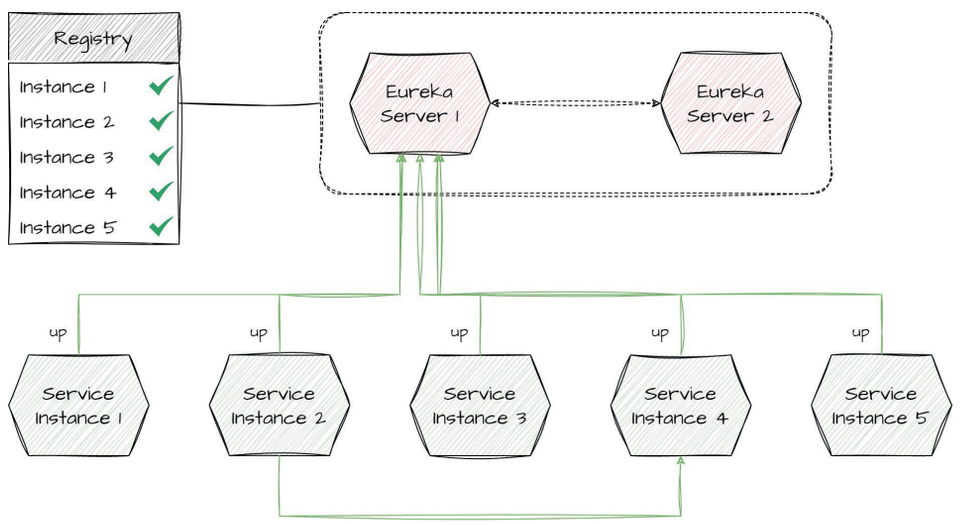
## Eureka clients register themselves with an Eureka server and look up other services.






Eureka does client-side service discovery. This means that a Eureka-enabled service integrates a Eureka client component that is responsible for registering the service instance at the Eureka server, but also for performing lookups against that Eureka server in case of an outgoing request to another Eureka-enabled service. Eureka uses logical names to identify services; the Eureka client requests the address of a service instance by resolving that logical name to the pubilc address of one of the service instances registered for that name. This allows for a round-robin communication scheme, since the Eureka server can flip through all service instances for subsequent requests.

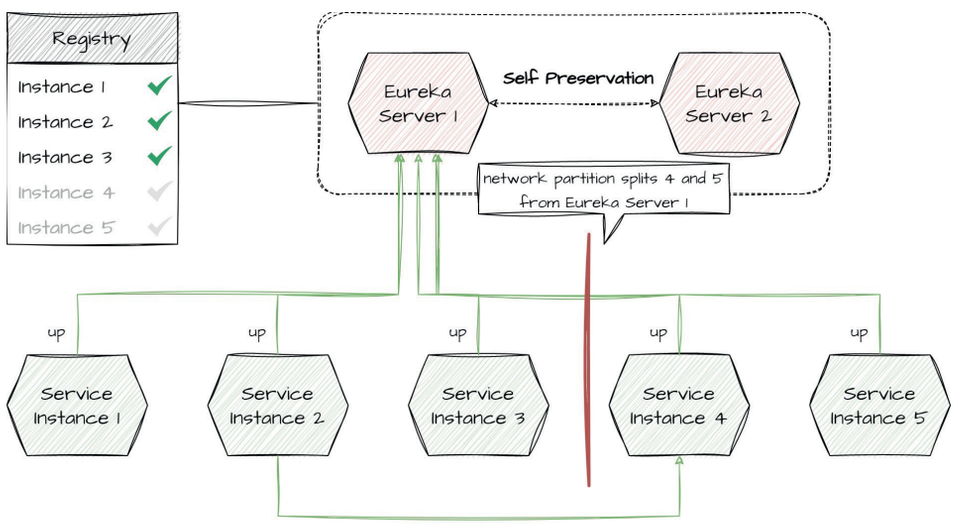
## Eureka clients send regular heartbeats to the Eureka server.






Eureka clients register with the first server configured in the `service-url` list. They only submit their heartbeat messages to this Eureka server as well. Suppose that all of the service instances are healthy and registered with Eureka server 1. Eureka server 1 replicates its state to its adjacent peers (Eureka server 2 in this example). Service instance 2 did a look up and as a result of that, has connectivity to service instance 4.

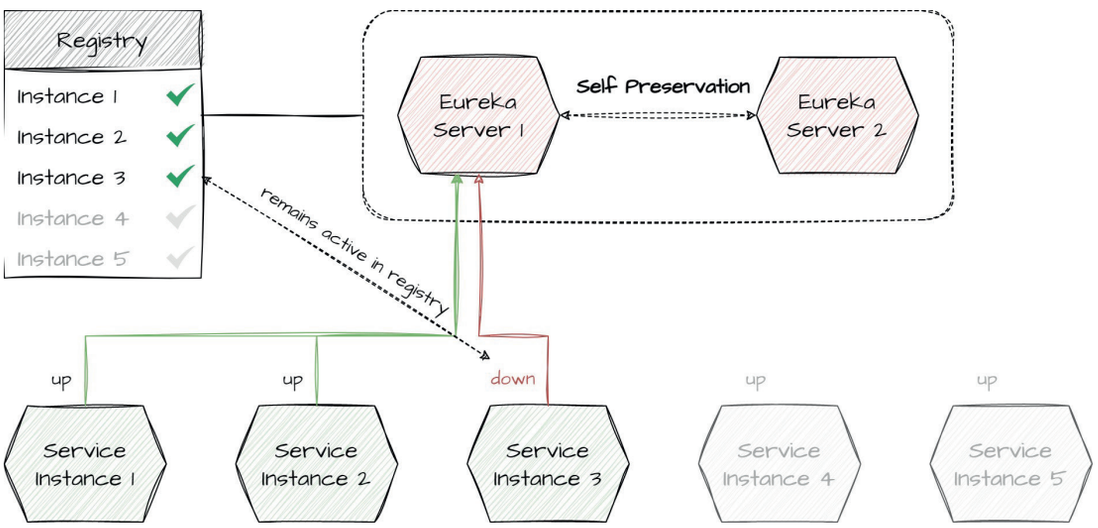
## Eureka servers enter self-preservation if actual heartbeats are below expectation.






Assume that a network partition splits service instances 4 and 5 from Eureka server 1. This means that Eureka server 1 will no longer receive heartbeat messages from these instances. Eureka server 1 will evict instance 4 and 5 from the registry, due to missing heartbeats. Service instance 2 is not affected by the partitions; it still has connectivity to service instance 4. Eureka server 1 observers that it lost more than 15% of the expected heartbeat messages and thus apply the self-preservation protocol. Eureka servers enter self-preservation mode if the actual number of heartbeats in the last minute is less than the expected number of heartbeats per minute. The expected number of heartbeats is tied to a interval of 30 seconds. Hence, every service instance is expected to submit 2 heartbeats per minute.

## During self-preservation, service instances won't get evicted.





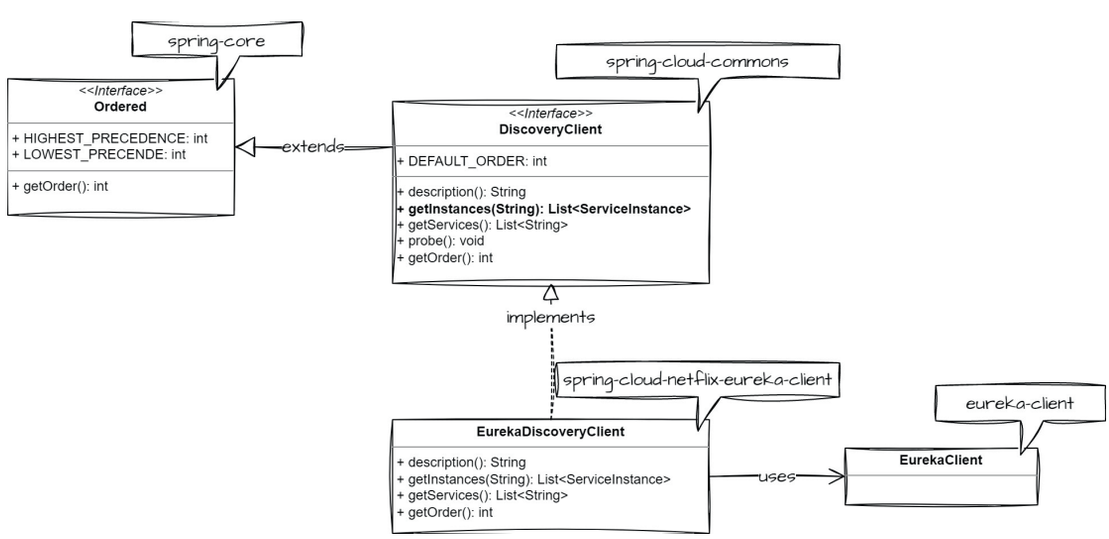

Eureka server 1 will stop evicting service instances if they go down from this point onward. Service instance 3 went down, but it remains active in the registry. Both Eureka servers would accept new registrations, though (self-preservation does not affect that).

## Spring Cloud offers an abstraction to communicate with discovery services.

- `DiscoveryClient` is central to the integration of discovery services.
- Spring Boot finds implementations for it during startup.
- Integrations available for
  - Netflix Eureka
  - Apache ZooKeeper
  - HashiCorp Consul



## `DiscoveryClient` is used to interact with discovery services.



**Spring Cloud offers an abstraction to load balance requests to registered instances.**

- Uses `LoadBalancerClient` to load balance to registered instances.
- Standard reactive `WebClient` is applicable for this

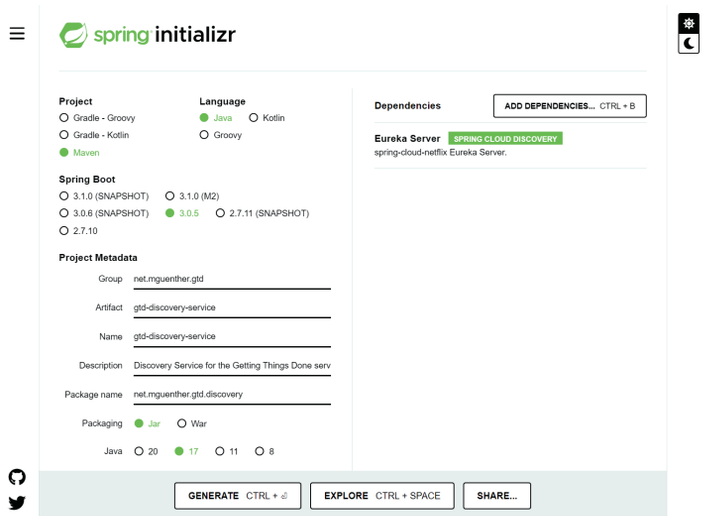
```
@Bean
@LoadBalanced
public WebClient.Builder loadBalancedWebClientBuilder() {
  return WebClient.builder();
}
```



Spring Cloud offers an abstraction to load balance requests to registered service instances via the `LoadBalancerClient` interface. Notably, a `LoadBalancerClient` integrates well with `DiscoveryClient`. The `LoadBalancerClient` consumes `ServiceInstance` objects and directs requests as appropriate. The standard reactive HTTP client, `WebClient`, can be configured to use an implementation `LoadBalancerClient`. The `LoadBalancerClient` is injected as an `ExchangeFilterFunction` into the `WebClient.Builder`.

## Setting up an Eureka server

## The Spring Initializr can be used to create a standalone Eureka server application.






Go to start.spring.io and select the shown dependencies.

## Adding @EnableEurekaServer turns the appplication into a fully fledged Eureka server.

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }
}
```



## Netflix Eureka offers a wide range of properties to configure the Eureka server.

- Documentation
  - `com.netflix.eureka.EurekaServerConfig`
- Default values
  - `org.springframework.cloud.netflix.eureka.server.EurekaServerConfigBean`

## Configure the discovery service for local development.

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      default-zone: http://`${eureka.instance.hostname}:$`{server.port}/eureka/
  server:
    wait-time-in-ms-when-sync-empty: 1000
    response-cache-update-interval-ms: 5000
```



Both the `eureka.instance` and `eureka.client` sections use the standard configuration for a standalone Eureka server. However, the `eureka.server` part in this example overrides a couple of default values. This is due to the fact that we want to minimize startup delays in a local development environment. For a production grade setup, you should go with service defaults first and change them as appropriate based on actual collected evidence.

## Configure the discovery service for containerized deployment.

- Override `eureka.instance.hostname` as appropriate

```
discovery-service:
  image: getting-things-done/discovery-service
  mem_limit: 256m
  ports:
    - "8761:8761"
  environment:
    eureka.instance.hostname: discovery-service
```



# Integrating an Eureka client

## Add the dependency to `spring-cloud-starter-netflix-eureka-client`.

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```



## Netflix Eureka offers a wide range of properties to configure the Eureka server.

Have a look at class

- `org.springframework.cloud.netflix.eureka.EurekaClientConfigBean`

for both

- documentation
- default values

## Eureka-enabled applications register themselves with a logical name.

- Eureka Client uses the value of `spring.application.name`

```
spring:
  application:
    name: logical-name-of-my-application
```



## Configure the Spring application for local development.

- Eureka client needs to know where to reach the Eureka server.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

## Disable the Eureka client for locally running `@SpringBootTests`.

- Tests annotated with `@SpringBootTest` require the application context
- They will likely fail locally (missing Eureka server instance)
- Prevent this with

```
@SpringBootTest(
  webEnvironment=RANDOM_PORT,
  properties={"eureka.client.enabled=false"}
)
```



## Configure the Spring application for containerized deployment.

- Override `EUREKA_CLIENT_SERVICEURL_DEFAULTZONE` with the proper value

```
query-service:
  image: getting-things-done/query-service
  mem_limit: 256m
  environment:
    SERVER_PORT: 8090
    SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
    EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: http://discovery-service:8761/eureka/
```

## Use the `WebClient` together with the logical name of the target service.

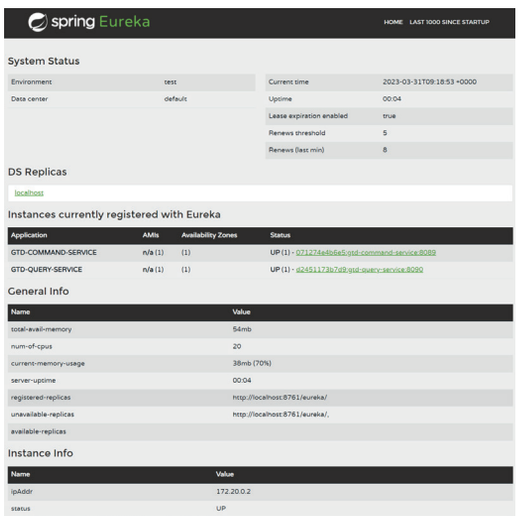
```
@Service
public class MyService {

  private static final String OTHER_SERVICE_URL = "http://other-service";

  private final WebClient webClient;

  @Autowired
  public MyService(WebClient.Builder webClientBuilder) {
    webClient = webClientBuilder.build();
  }
}
```

- `other-service` is the logical name of the target service
- as defined by its `spring.application.name` parameter



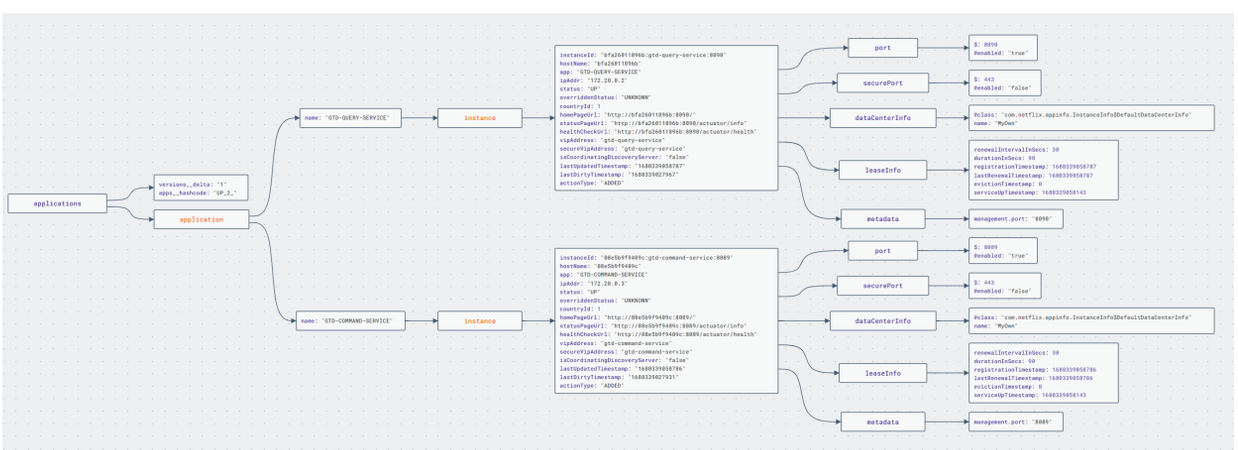
# Retrieving information from Eureka server

## The Eureka server dashboard holds information on registered instances.






## The Eureka server exposes a HTTP API that gives detailed information.



# Going to production

**A production-grade deployment requires high availability.**

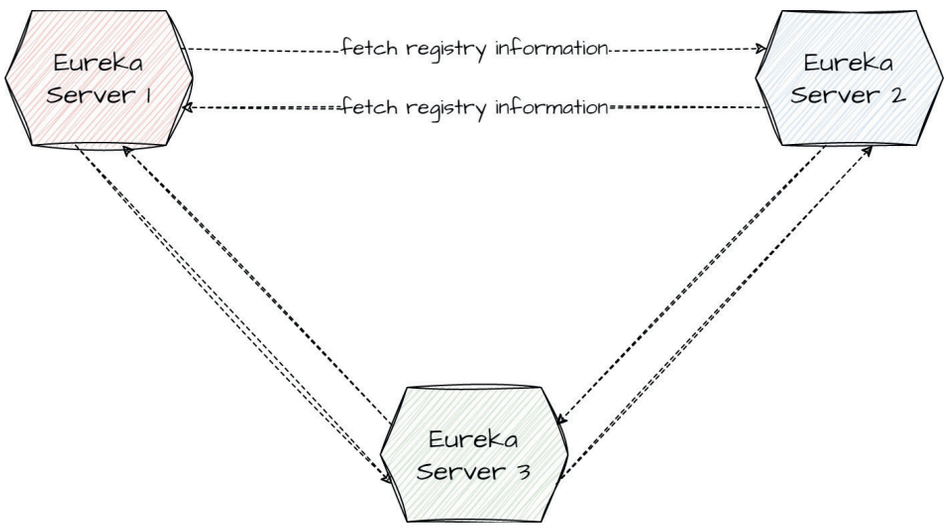
**Eureka achieves high availability at two levels**

- Server Cluster
- Client Side Caching



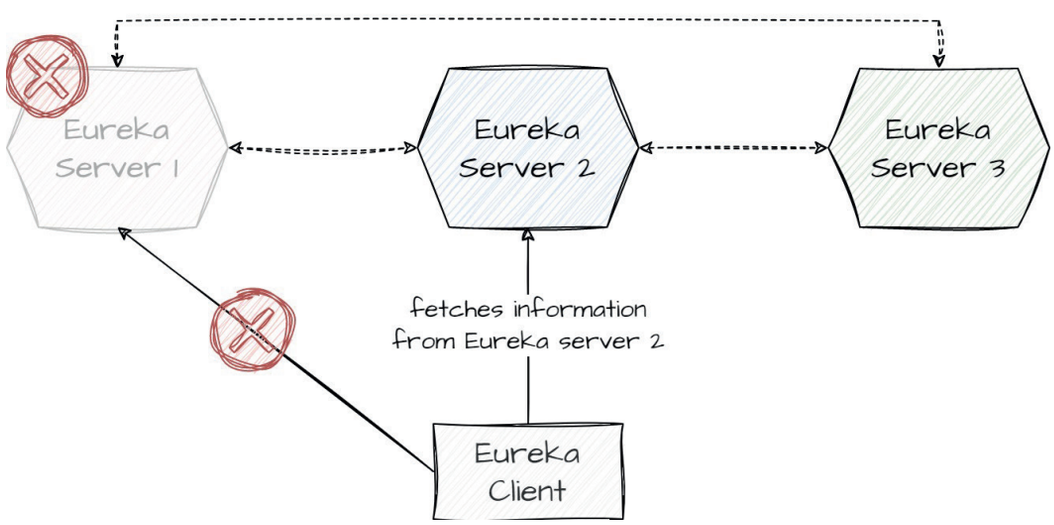
Eureka can be deployed as a cluster of servers. In case one of these servers crash, clients can still connect to the remaining Eureka servers and discover services. Clients retrieve and cache registry information from an Eureka server. In case all servers crash, clients still posses the last healthy snapshot of the registry. This is the default behavior of Eureka clients. There is no additional configuration required to enable client side caching.

## Eureka server instances register and synchronize with each other.






## If one server goes down, the Eureka client switches to another.



## Enabling peer awareness is only a matter of configuration.

```
spring:
  profiles: peer-1
  application:
    name: eureka-server-clustered
server:
  port: 9001
eureka:
  instance:
    hostname: peer-1-server.com
client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
    defaultZone: http://peer-2-server.com:9002/eureka/
```



## Enabling peer awareness is only a matter of configuration. (cont.)

```
spring:
  profiles: peer-2
  application:
    name: eureka-server-clustered
server:
  port: 9002
eureka:
  instance:
    hostname: peer-2-server.com
client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
    defaultZone: http://peer-1-server.com:9002/eureka/
```

## Eureka clients should be configured to use a list of Eureka server instances.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer-1-server.com:9001/eureka/, http://peer-2-server.com:9002/eureka/
```

**Check if self preservation works for your environment.**

- Can lead to false positives
- Never expires, unless
  - ... network partition heals itself
  - ... services instances are brought back up
- Unable to fine-tune heartbeat intervals if enabled



Under certain network conditions self preservation mode may lead to a significant number of false-positives, where Eureka incorrectly assumes that service instances are down - but they are not. Also, if enabled, you're unable to fine-tune heartbeat intervals because self preservation always assumes that heartbeats are sent within intervals of 30 seconds. The calculation of the expected heartbeats per minute relies on these 30 seconds.

# Advanced Spring

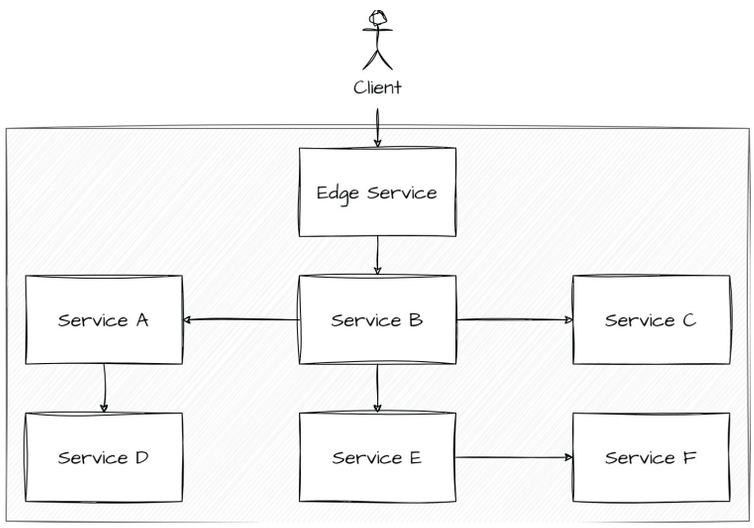
## Edge Service with Spring Cloud Gateway

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

## What is an edge service and how does it fit into Cloud-based architectures?






Edge services behave like reverse proxies. They can be integrated with a discovery service. This provides dynamic load balancing capabilities.

**What are typical use cases for edge services?**

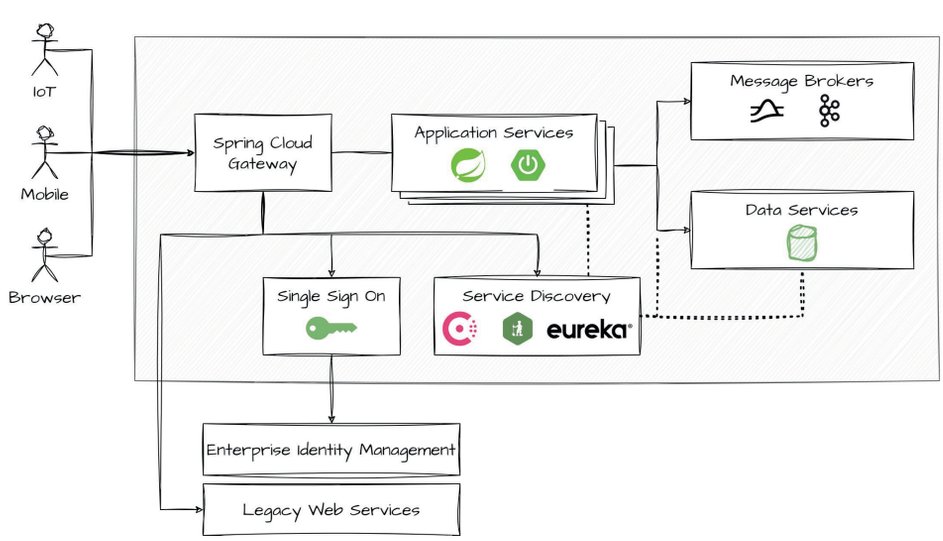
- Hide internal services
- Expose external services
- Protect services from malicious requests
- Integrate legacy services



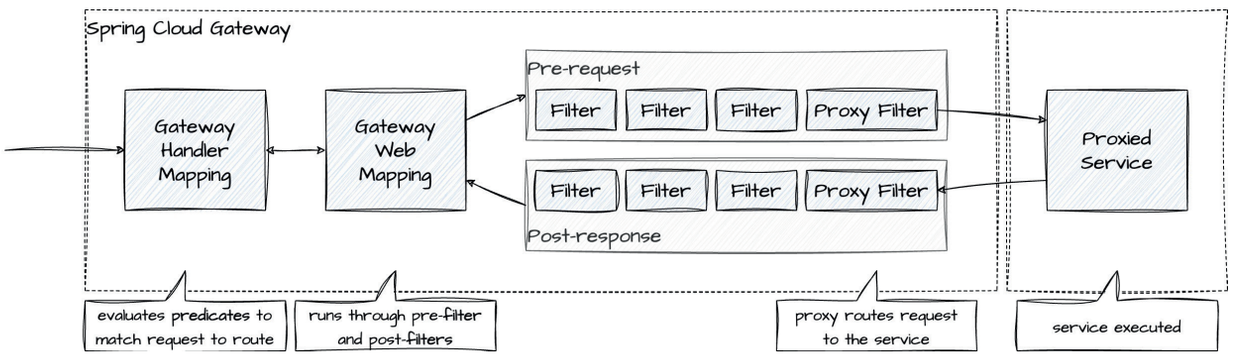
Use standard protocols and best practices such as OAuth, OIDC, JWT and API keys to ensure that clients are trustworthy.

# Spring Cloud Gateway

**Spring Cloud Gateway is a battle-proven edge service that integrates many other services.**



## Spring Cloud Gateway is built around gateway, routes, and filters.






The **gateway** is the core component of Spring Cloud Gateway. It acts as an entry point for incoming HTTP requests. It is responsible for handling the request-routing process, which includes matching requests to appropriate routes and applying filters.

A **route** is also a fundamental element of Spring Cloud Gateway. A **route** defines a mapping between a request and a destination.

**Predicates** are conditions that determine if an incoming request matches a specific route. Spring Cloud Gateway provides built-in predicates, such as `Path`, `Host`, and `Method`, but it is also possible to create custom predicates.

**Filters** are responsible for modifying requests and responses during the routing process. Spring Cloud Gateway provides a range of built-in filters, like `AddRequestHeader`, `RewritePath`, and `SetStatus`, but it is also possible to implement custom filters.

**Routes are the core concept that enables the gateway to forward incoming requests.**

**Routes are composed of**

- **ID**: Unique identifier for the route
- **Predicates**: Conditions that must be met to match a request
- **Filters**: List of pre-request and post-response processors
- **URI**: Target URI to which the request will be forwarded

**Routes can either be configured using Java or YAML code.**

**Benefits of the Java-based configuration**

- Dynamic route configuration
- Advanced filtering and custom logic
- Programmatic access to Spring components
- Strong type checking



(1) When you need to create routes dynamically based on runtime conditions, such as configuration values fetched from a database or an external configuration service, the Java-based approach is more suitable.

(2) If your route configuration requires advanced filters or custom logic that goes beyond the capabilities of the built-in filters and predicates, the Java-based allows you to easily reference and use custom classes in the configuration.

(3) Easier access to beans.

(4) Strong type checking is especially helpful during development.

## Routes can either be configured using Java or YAML code. (cont.)

**Benefits of the YAML-based configuration**

- Simplicity and readability
- External configuration
- Reduced code complexity
- Easier collaboration
- Consistent configuration format



(1) YAML-based route configuration provides a more concise and declarative way to define routes.

(2) The YAML-based approach allows you to store route configurations in external files, making it easy to manage and version the configuration independently of your application code.

(3) Reduces the amount of code required to write and maintain for route configurations.

(4) More approachable for developers of different levels of experience and backgrounds.

(5) If you're already using YAML, the YAML-based approach provides a consistent format across the entire application.

## `RouteLocatorBuilder` provides an easy-to-use API to configure routes.

```java
@Bean
public RouteLocator myRoute(RouteLocatorBuilder builder) {
  return builder
    .routes()
    .route("my-route-id",
                // use the path predicate
           r -> r.path("/example/**")
                // apply a pre-request filter
                .filters(f -> f.addRequestHeader("My-Header", "My-Value"))
                // configure the target URI
                .uri("http://localhost:8080"))
    .build();
}
```



Use the `RouteLocatorBuilder` to create beans of type `RouteLocator`. The `RouteLocator` defines the route with the specified components. When a request comes in that matches the predicate (in this case it must match a path beginning with `/example`), the filter will be applied, before the modified request will be forwarded to the target URI. `RouteLocatorBuilder` also allows you to chain multiple route definitions together. This makes it easy to define and manage multiple routes in your gateway configuration.

**The same route can also be expressed using a YAML-based configuration.**

```
spring:
  cloud:
    gateway:
      routes:
      - id: my-route-id
        uri: http://localhost:8080
        predicates:
        - Path=/example/**
        filters:
        - AddRequestHeader=My-Header, My-Value
```



As discussed, the YAML-based configuration is somewhat less powerful than the Java approach. But when it comes to simplicity and good readability, a concise YAML configuration for a static might just be what you need.

## So, which one do you prefer ... ?

It is perfectly fine to mix-and-match based on the given situation!

**Spring Cloud Gateway come with several built-in predicates.**

1. on path segments: `Path=/api/**`
2. on domain names: `Host=*.example.com`
3. on headers: `Header=X-Request-Type, Mobile`
4. on HTTP method: `Method=POST`
5. on query parameters: `Query=version, 1.0`
6. on client's IP address: `RemoteAddr=192.168.1.0/24`

Multiple predicates are chained using the AND operator



(1) This predicate matches requests based on the request path. It supports path patterns using wildcard characters. The example matches any request for which the path starts with `/api/`.

(2) This predicate matches requests based on the request host (domain name). It also supports wildcard patterns. The example matches any request with a host ending in `.example.com`.

(3) This predicates matches requests based on a particular header name and value.

(4) This predicates matches requests based on their HTTP method.

(5) This predicate matches requests based on the presence of a query parameter with a specific value.

(6) Matches based on the client's IP address.

**Spring Cloud Gateway makes it easy to add your own predicate logic.**

- A `GatewayPredicate` holds the logic for predicate evaluation
- A `RoutePredicateFactory` provides the means to
  - configure a `GatewayPredicate`
  - use a `GatewayPredicate` for rule evaluation
- Implement these interfaces to provide a custom predicate
- Annotate the factory with `@Component` or provide a Spring bean for it



Spring Cloud Gateway leverages the factory method pattern, not only for predicates, but also for filters. There are certain rules for predicate look up in place. Please consult the Spring Cloud Gateway documentation for further information on this.

**Spring Cloud Gateway comes with several built-in pre-request filters.**

1. `AddRequestHeader=X-Request-Source, Gateway`
2. `RemoveRequestHeader=X-Sensitive-Header`
3. `RewritePath=/api/v1/(?<segment>.*), /new-api/v1/$1`
4. `PrefixPath=/api`
5. `Retry(times=3, backoff.firstBackoff=100ms, backoff.maxBackoff=500ms, backoff.factor=2, status=500)`



(1) This filter adds a new header to the incoming request with a specified name and value.

(2) Thils filter removes a header from the incoming with a specified name.

(3) This filter rewrites the request path based on a regular expression and replacement pattern. In the example, we rewrite the path `/api/v1/users` to `/new-api/v1/users`.

(4) This filter adds a prefix to the request path. In this example, we change `/users` to `/api/users`.

(5) This filter enables retries for a route.

**... as well as several built-in post-request filters.**

1. `AddResponseHeader=X-Response-Generated-By, Gateway`
2. `RemoveResponseHeader=X-Internal-Header`



(1) This filter adds a new header to the outgoing response with a specified name and value.

(2) This filter removes a header from the outgoing response with a specified name.

**Rolling your own filter is a matter of implementing two interfaces.**

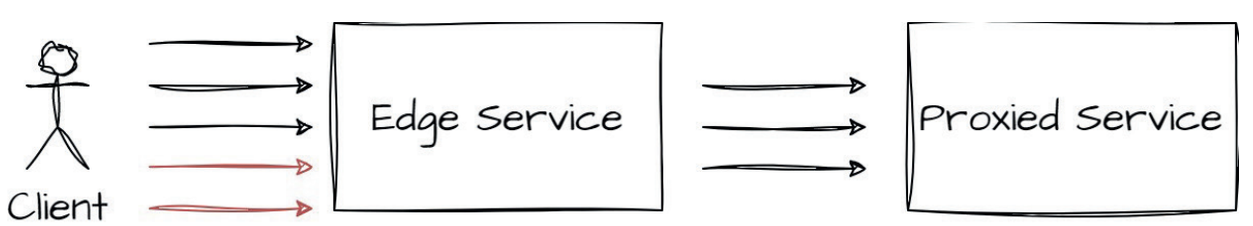
- A `GatewayFilter` holds the logic for filter evaluation
- A `GatewayFilterFactory` provides the means to
  - configure a `GatewayFilter`
  - use a `GatewayFilter` for applying the filter
- Implement these interfaces to provide a custom filter
- Annotate the factory with `@Component` or provide a Spring bean for it



This works in the same way as with predicates.

# Advanced Features and Use Cases

**Limit the rate of incoming requests to not overburden proxied services.**



**A rate limiting strategy can incorporate different kinds of request data.**

**Apply rate limiting on**

- any source request
- a unique set of clients based on their IP
- a unique set of user agents
- specific request parameters, such as API keys

**Spring Cloud Gateway offers built-in support for rate limiting.**

**`RequestRateLimiter` filter determines if a request**

- is allowed to proceed
- is denied due to limit exhaustion
- lets you use custom `RateLimiter` strategies
- lets you manage different services by key (`KeyResolver`)



Spring Cloud Gateway comes with a built-in `KeyResolver` that uses the `Principal` of a user. A secured gateway is required to resolve the principal, though. But you also have the option to implement the `KeyResolver` interface.

## A `RateLimiter` determines whether requests are allowed to proceed.

```
public interface RateLimiter<C> extends StatefulConfigurable<C> {
  Mono<Response> isAllowed(String routeId, String id);

  class Response {
    private final boolean isAllowed;
    private final long tokensRemaining;
    private final Map<String, String> headers;
    /* constructor and getters omitted */
  }
}
```

## A `KeyResolver` is responsible for determining the key used for rate limiting.

```
public interface KeyResolver {
  Mono<String> resolve(ServerWebExchange exchange);
}
```

- Rate limit key is used to track # of requests by specific client(s)
- Enforce rate limits for different sets of clients



Spring Cloud Gateway comes with a built-in strategy that is able to resolve the user's principal. A secured gateway is required though, to extract this information from the `ServerWebExchange`. You can roll your own strategy by implementing the `KeyResolver` interface. Common key resolution strategies include using a client's IP address, user ID, or API key.

## Example: Use a fixed interval rate limiter (not for production use!)

```
public class FixedIntervalRateLimiter implements RateLimiter<Config> {
  private final Duration duration;
  private final int limit;
  private final ConcurrentHashMap<String, AtomicInteger> counters;
  /* constructors and such omitted */
  @Override
  public Mono<Response> isAllowed(String routeId, String id) {
    var counter = counters.computeIfAbsent(id, key -> new AtomicInteger(0));
    if (counter.incrementAndGet() <= limit) {
      return Mono.just(new Response(true, Map.of()));
    } else {
      if (counter.get() == limit + 1) {
        Mono.delay(duration).doOnTerminate(() -> counters.remove(id)).subscribe();
      }
      return Mono.just(new Response(false, Map.of());
    }
  }
}
```

## Example: Apply rate limiting for individual clients based on their IP addresses.

```
public class IpAddressKeyResolver implements KeyResolver {
  @Override
  public Mono<String> resolve(ServerWebExchange exchange) {
    var ipAddress = exchange.getRequest()
      .getRemoteAddress()
      .getAddress()
      .getHostAddress();
    return Mono.just(ipAddress);
  }
}
```



In this example, the `IpAddressKeyResolver` extracts the client's IP address from the incoming request and uses it as the rate limit key. This ensures that rate limiting is applied on a per-client basis, where each client is identified by their IP address.

## Example: Configure beans for the custom `RateLimiter` and custom `KeyResolver`.

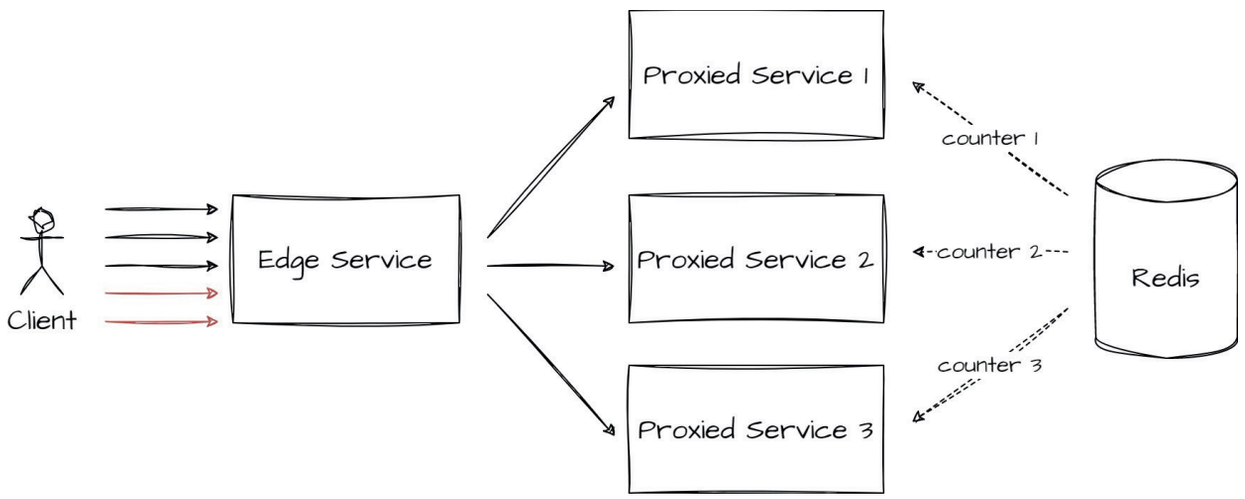
```
@Configuration
public class GatewayConfiguration {
  @Bean(name = "fixedIntervalRateLimiter")
  public FixedIntervalRateLimiter fixedIntervalRateLimiter() {
     return new FixedIntervalRateLimiter(Duration.ofSeconds(5), 1);
  }

  @Bean(name = "ipAddressKeyResolver")
  public KeyResolver ipAddressKeyResolver() {
     return new IpAddressKeyResolver();
  }
}
```



## Example: Use the `RateLimiter` and `KeyResolver` on a dedicated route.

```
spring:
  cloud:
    gateway:
      routes:
        - id: query-service
          uri: http://www.example.org
          predicates:
            - Path=/**
          filters:
            - name: RequestRateLimiter
              args:
                rate-limiter: "#{@fixedIntervalRateLimiter}"
                key-resolver: "#{@ipAddressKeyResolver}"
```

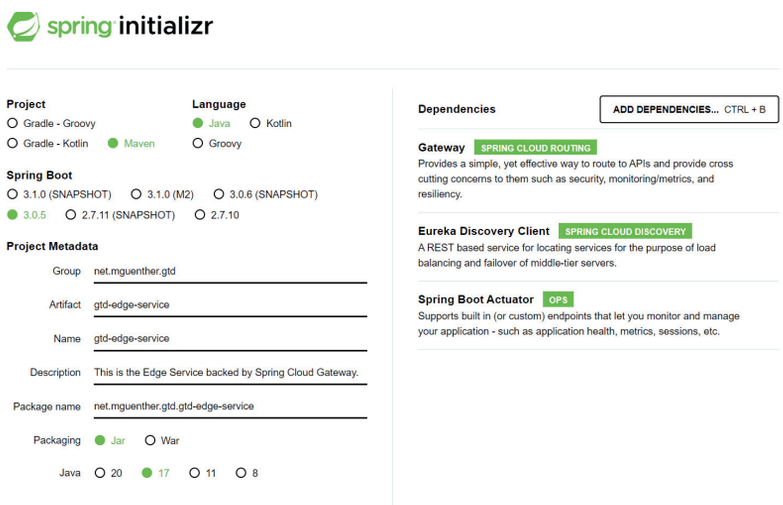
A production-grade rate limiting strategy requires distributed counters.






# Setting up an Edge Service with Spring Cloud Gateway

## The Spring Initializr can be used to create a standalone edge service.






## Provide the bean for a load-balancer aware `WebClient`.

```
@SpringBootApplication
public class GatewayApplication {

  public static void main(String[] args) {
    SpringApplication.run(GatewayApplication.class, args);
  }

  @Bean
  @LoadBalanced
  public WebClient.Builder loadBalancedWebClientBuilder() {
    return WebClient.builder();
  }
}
```

## Configure Spring Boot Actuator for development usage.

```
management:
  endpoint:
    gateway:
      enabled: true
    health:
      show-details: "ALWAYS"
  endpoints:
    web:
      exposure:
        include: "*"
```



## Add a composite health check for all services that the gateway routes to.

```
@Configuration
public class HealthCheckConfiguration {

  @Bean
  public ReactiveHealthContributor healthcheckServices(@Autowired WebClient.Builder builder) {

    var webClient = builder.build();
    var registry = new LinkedHashMap<String, ReactiveHealthIndicator>();

    registry.put("command-service", () -> determineHealth("http://gtd-command-service", webClient));
    registry.put("query-service", () -> determineHealth("http://gtd-query-service", webClient));

    return CompositeReactiveHealthContributor.fromMap(registry);
  }

  private Mono<Health> determineHealth(final String baseUrl, final WebClient webClient) {
    var url = baseUrl + "/actuator/health";

    return webClient.get().uri(url).retrieve()
      .bodyToMono(String.class)
      .map(s -> new Health.Builder().up().build())
      .onErrorResume(ex -> Mono.just(new Health.Builder().down(ex).build()))
      .log();
  }
}
```

## Add a composite health check for all services that the gateway routes to. (cont.)

This adds the following section to the output of `/actuator/health`

```
"healthcheckServices": {
    "status": "UP",
    "components": {
        "command-service": {
            "status": "UP"
        },
        "query-service": {
            "status": "UP"
        }
    }
}
```

# Advanced Spring

## Spring Cloud Config

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

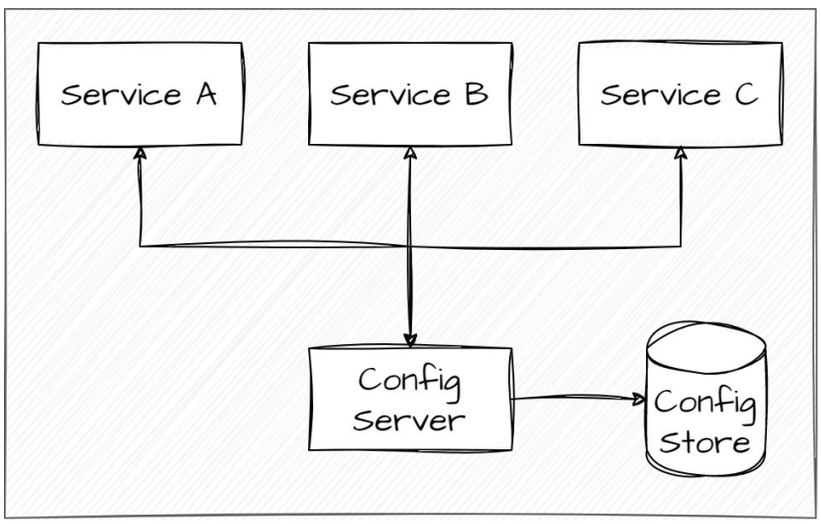
Mastichari, Kos

**Traditional applications are deployed together with their configuration.**

**This bears some questions**

- How do I get a complete picture of the configuration?
- How do I update the configuration for all service instances?
- How do I share common parts of the configuration between services?

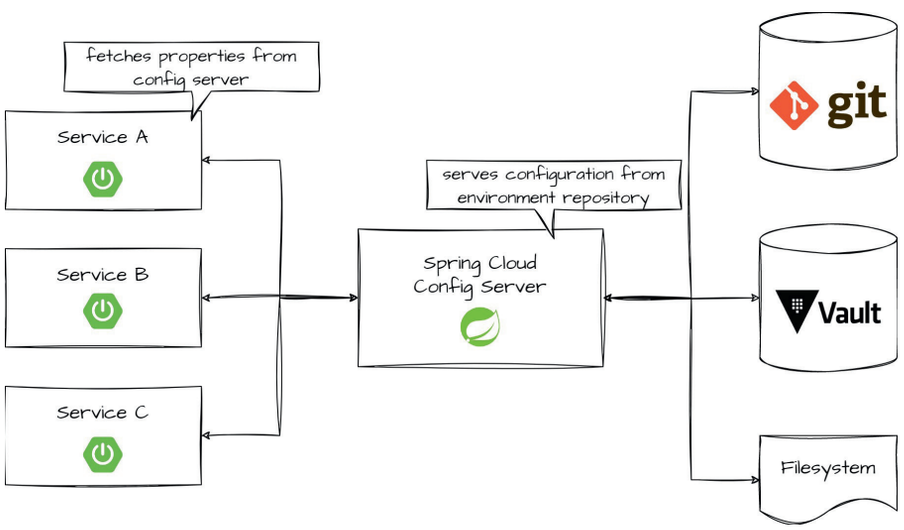
A config server simplifies centralized configuration management and promotes consistency.






# Spring Cloud Config Server

**Services and config server exchange configuration data via an HTTP API.**



**The Config Server HTTP API provides the means to access configuration data dependent on the target environment.**

**Endpoint patterns**

- `/{application}/{profile}`
- `/{application}/{profile}/{label}`
- `/{application}/{profile}/{label}/{path}`



(1) Yields the configuration for the specified `application`, where `application` is the name as defined by property `spring.application.name`. The `profile` must be set as well. This is either `default` for common, profile-independent configuration data, or refers to a dedicated profile. On the client side, this is essentially the value bound to `spring.profiles.active`.

Options (2) and (3) apply if we use a VCS like Git as storage backend for the Spring Cloud Config Server. `label` then refers to either a commit ID, branch name, or a tag. `path` is a path to a filename (e.g. `log.xml`) in the VCS.

**Properties `spring.application.name` and `spring.profiles.active` determine which configuration should be fetched.**

```
{
  "name": "gtd-command-service",
  "profiles": ["default"],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
      "name": "file:/opt/app/config/gtd-command-service.yml",
      "source": {
      "server.port": 8089,
      "server.servlet.context-path": "/",
      "server.forward-headers-strategy": "framework",
      "eureka.client.serviceUrl.defaultZone": "http://localhost:8761/eureka",
      "management.security.enabled": false,
      "spring.application.name": "gtd-command-service",
      "spring.kafka.bootstrap-servers": "localhost:9092",
      "spring.kafka.consumer.value-deserializer": "net.mguenther.gtd.event.ItemEventDeserializer",
      [...]
}}]}
```

**The HTTP API also exposes additional management endpoints.**

**Endpoints**

- `/actuator`
- `/encrypt`
- `/decrypt`
- `/{encrypt|decrypt}/status`



(1) This is the standard actuator endpoint. Use it with care! Actuator endpoints are a useful tool for debugging a service during development, but should be locked down in a production-grade setup.

(2, 3) This is the endpoint for encrypting and decrypting sensitive information. These endpoints also pose a security risk and must be locked down in a production-grade setup. Use the /encrypt endpoint to create encrypted values that you can place in your property file. Use the /decrypt endpoint to verify that the encrypted information is indeed correct.

(4) Both endpoints for encrypting and decrypting sensitive information also feature a status endpoint that can be used to get some information on the availability on the encryption feature.

## Use the encrypt endpoint to create encrypted values.

```
curl --request POST \
     --url http://localhost:8888/encrypt \
     --header 'Authorization: Basic YWRtaW46YWRtaW4=' \
     --header 'Content-Type: application/x-www-form-urlencoded' \
     --data 'hello world'

b5af34ce8a920eab1131bc82b485ed9f488f51fc17d38113147d762ba3a91780
```



## Use the decrypt endpoint to verify encrypted information.

```
curl --request POST \
  --url http://localhost:8888/decrypt \
  --header 'Authorization: Basic YWRtaW46YWRtaW4=' \
  --header 'Content-Type: application/x-www-form-urlencoded' \
  --data b5af34ce8a920eab1131bc82b485ed9f488f51fc17d38113147d762ba3a91780

hello world
```

**Spring Cloud Config Server supports a variety of storage backends.**

These are called **environment repositories**:

- Local filesystem
- Git repository
- HashiCorp Vault
- JDBC database

full list available at https://docs.spring.io/spring-cloud-config/docs/current/reference/html/#_environment_repository

## An environment repository backed by a local filesystem has a flat structure ...

```
config-repository/
|-- application.yaml
|-- edge-service.yaml
|-- eureka-server.yaml
|-- service-a.yaml
|-- service-b.yaml
```



The common parts have been placed in a common configuration file, call it `application.yaml`. This file is shared by all clients. All other filenames reflect the name of the application as configured per property `spring.application.name`.

## ... in which YAML-files comprise multiple environment configurations.

```
<properties of the default profile>

---
spring.config.activate.on-profile: docker

<properties of the 'docker' profile>

---
spring.config.activate.on-profile: test

<properties of the 'test' profile>
```



## What about security concerns?

*Configuration properties should be treated as sensitive data.*

**Secure the configuration in transit.**

- Don't expose the config server via the edge service
- Use HTTPS for accessing the API
- Example: HTTP Basic Authentication with Spring Security
  - `SPRING_SECURITY_USER_NAME`
  - `SPRING_SECURITY_USER_PASSWORD`

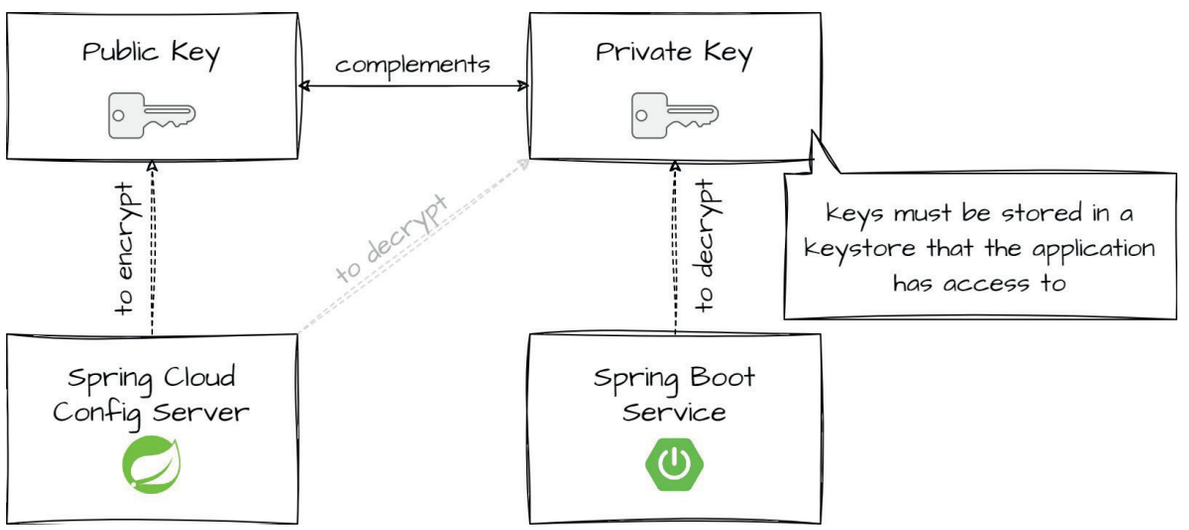
**Secure the configuration at rest.**

- Config Server supports the encryption of configuration properties
- Both symmetric and asymmetric keys are admissible
- Example: Activate a symmetric key via the environment
  - ENCRYPT_KEY



Configuration properties should be secured at rest as well. This is to avoid a scenario in which someone with access to the environment repository is able to steal sensitive information. The Spring Cloud Config Server supports the encryption and decryption of configuration properties by means of symmetric and asymmetric keys. Asymmetric keys are a bit harder to manage as symmetric keys, but they also offer a stronger security. Asymmetric keys rely on a keystore.

**Asymmetric keys are a harder to manage, but offer stronger security.**






Spring Cloud Config Server offers two ways to operate with a public-/private-key-pair.

(1) Present the public key to the config server and leverage its encryption capabilities. Encrypted values must be stored with the `{cipher}` prefix at rest. Being unable to decrypt the cipher, the config server returns the encrypted values to the client-side service. The client-side service knows about the private key and is able to decrypt the value.

(2) Present both the public and private key to the config server. In this case, the config server is able to decrypt encrypted config values and return the decrypted config value to the client-side service. Although this is possible, following along this path bears no significant value over symmetric keys, but increases the management overhead that comes with using public-/private-key-pairs.

# Spring Cloud Config Client

**Spring Cloud Config Client needs to be integrated with the Spring application.**

- Required dependencies
  - `spring-cloud-starter-config`
  - `spring-boot-starter-web`
  - `spring-retry`
- Move `application.yaml` to environment repository (and rename it)
- Configure access to the config server (`application.yaml` in service)
- Disable the config server in tests that require Spring context

**Processing of multiple property files has changed with Spring Boot 2.4.0.**

- Redefined the order in which property files are loaded
- Properties declared lower in a file will override those higher up
- Use `spring.config.import` to load additional property files
- Since Spring Cloud Config 3.0.0 (part of Spring Cloud 2020.0.0)
- No need for legacy `bootstrap.yaml` files

## Configure access to the config server. Use a retry policy to mitigate transient errors.

```yaml
spring.config.import: "optional:configserver:"

spring:
  application:
    name: gtd-command-service
  cloud:
    config:
      fail-fast: true
      retry:
        initial-interval: 5000
        multiplier: 1.3
        max-interval: 10000
        max-attempts: 5
      uri: ${CONFIG_SERVER_URI:http://localhost:8888}
      username: ${CONFIG_SERVER_USERNAME}
      password: ${CONFIG_SERVER_PASSWORD}
```



(1) It is possible to specify the complete URI at which the config server is listening in `spring.config.import`. If we omit this information, Spring Cloud Config Client uses `spring.cloud.config.uri`. This enables us to override the URI using environment variables. The `optional` prefix states that the application startup should not fail in case the application was unable to fetch the configuration from the config server. If we omit this prefix and simply use `configserver:` as value for `spring.config.import`, the application will fail at startup if the config server is not reachable.

(2) In this example, we're using HTTP Basic Authentication as a means to secure the access to the configure. Hence, we have to configure `spring.cloud.config.username` as well as `spring.cloud.config.password` with the appropriate user credentials.

(3) The retry mechanism is configured as follows: Retry at most 5 times (cf. `spring.cloud.config.retry.max-attempts`), wait for an initial 5000 ms if the first request does not succeed (cf. `spring.cloud.config.retry.initial-interval`), before trying it again. Apply exponential back-off by increasing the waiting period between attempts by a factor of 1.3 (cf. `spring.cloud.config.retry.multiplier`). The maximum wait time between retries is configured 10000 ms (cf. `spring.cloud.config.retry.max-interval`).

(4) If Spring Cloud Config Client is not able to fetch the configuration within the boundaries of these retry semantics, application startup will fail.

## Configuring the client for local development can get tricky ...

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, properties = {
  "eureka.client.enabled=false",
  "spring.cloud.config.discovery.enabled=false",
  "spring.cloud.config.enabled=false"
})
class GtdCommandServiceApplicationTests {

  @Test
  void contextLoads() {
  }
}
```

- No dependency on config server or service discovery locally
- Breaks application context

```
java.lang.IllegalStateException: Unable to load config data from 'optional:configserver:'
```



This is due to the fact that Spring Cloud Config Client tries to parse the property value of `spring.config.import`, sees that the prefix is admissible, but omits any further action because `spring.cloud.config.enabled=false`. Thus, instead of the `ConfigServerConfigDataLocationResolver`, the `StandardConfigDataLocationResolver` is used as a fallback to evaluate the property value of `spring.config.import`. This fails, as it cannot make any sense of the provided value.

**... but there is workaround for it!**

- Move config server configuration to dedicated profile
- Use the default profile for configuring the service for local development
- Lauch the application with profile `configserver` activated
- Run tests etc. using the `default` profile



## Still, there is another problem with this configuration!

## What about the value of `spring.cloud.config.uri`?

**Spring Cloud Config can be integrated with a discovery service, such as Netflix Eureka.**

- Register the Spring Cloud Config Server with the discovery service
- Configure the `DiscoveryClient` on the client-side
  - `spring.cloud.config.discovery.enabled=true` (defaults to `false`)
  - `spring.cloud.config.discovery.service-id=<config-server-id>`
  - this is called *Discovery First Bootstrap*
- Configure Netflix Eureka on the client-side

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery-service:8761/eureka
```

## Spring Boot disallows profile-specific properties when using Discovery First Bootstrap.

- Configure service discovery in the default profile
  - Won't work locally without Eureka, but since we disable it anyways ...
  - Use it as a basis for specific profiles

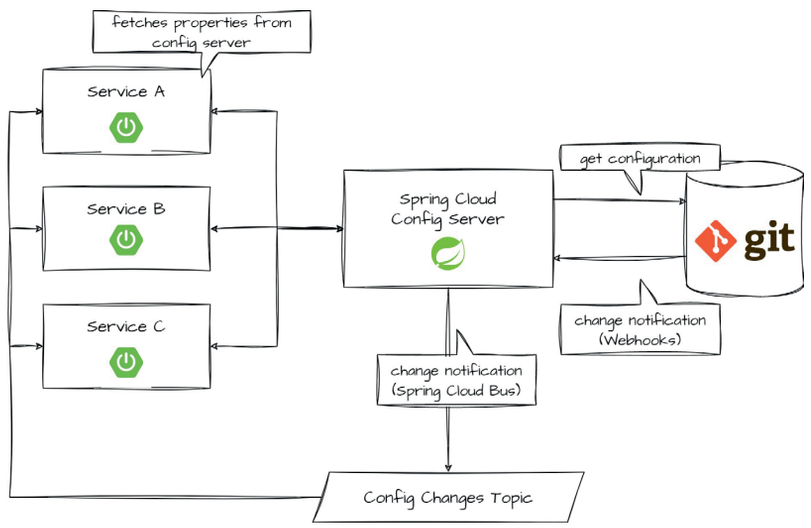
```yaml
spring:
  cloud:
    config:
      discovery:
        enabled: true
        service-id: config-server
```



## Get rid of `spring.cloud.config.uri` in the `configserver` profile.

```yaml
spring.config.activate.on-profile: configserver
spring.config.import: "optional:configserver:"

spring:
  application:
    name: gtd-command-service
  cloud:
    config:
      fail-fast: true
      retry:
        initial-interval: 500
        multiplier: 1.3
        max-interval: 1000
        max-attempts: 5
      username: ${CONFIG_SERVER_USERNAME}
      password: ${CONFIG_SERVER_PASSWORD}
```

## Integrate Spring Cloud Bus to support configuration updates on running service instances.






(1) As soon as service instances start up, they fetch their configuration from the config server.

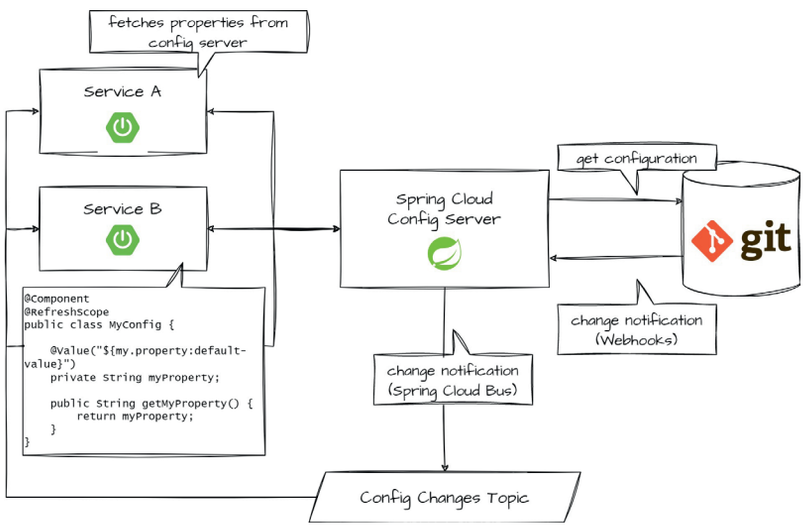
(2) The config server retrieves the configuration from a Git repository.

(3) The Git repository is configured to publish notifications on certain events, e.g. commits that are pushed to the Git repository. This is done using Webhooks.

(4) The config server observers these events and uses Spring Cloud Bus to publish change notifications to a dedicated topic. There is out-of-the-box support for Apache Kafka and RabbitMQ.

(5) The service instances react if they are affected and retrieve the updated configuration from the config server.

## Use the @RefreshScope annotation to ensure bean re-creation on refresh events.






## Takeaway

- Centralized configuration management simplifies and promotes consistency
- Config Server serves properties
- Config Client integrates with Spring Boot applications
- Supports handling of sensitive configuration data
- Dynamic configuration refresh enables updates and consistent configurations

# Advanced Spring

## Introduction to Resilience4j

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### What is resilience?

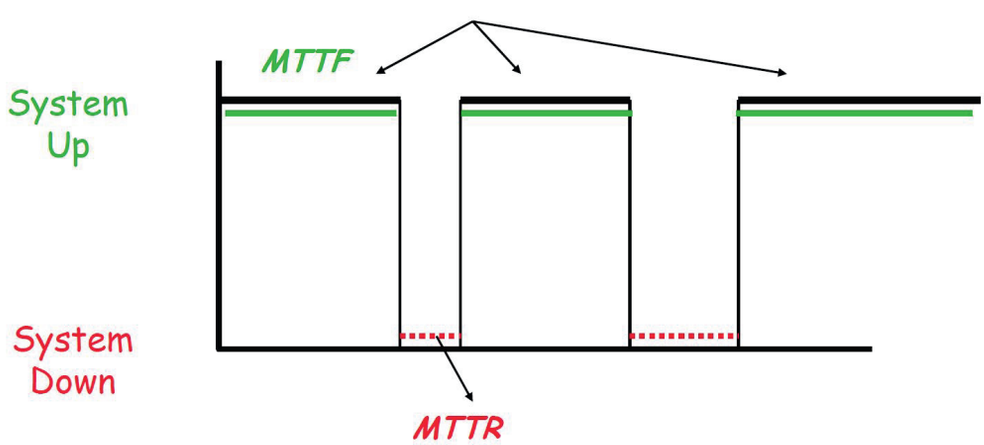
*Resilience (or fault tolerance) refers to the **property** of a (distributed) system to **maintain its functionality** even when **unforeseen** inputs or errors in hardware or software occur.*

**Resilient software design is concerned with aspects of robustness against errors.**

- A service might suffer from high response times.
- A service might respond incorrectly.
- A service might not respond at all.
- The infrastructure or service topology has changed.
- The system exhibits random, unexpected errors.



**Let's discuss availability.**




MTTF = Mean Time To Failure, MTTR = Mean Time To Recovery

**The availability metric sets MTTF and MTTR into relation.**

Availability = Total Up-Time / (Total Up-Time + Total Down-Time)

or

**Availability = MTTF / (MTTF + MTTR)**



Availability is defined as MTTF / (MTTF + MTTR), where MTTF is the *mean time to failure* and MTTR is the *mean time to recovery*. The traditional approach to stabilising a system of services is to increase MTTF. This is done by adding redundancy: more servers, more network connections, ...

## But what does it mean?

Availability = (365 days * 24 hours - 8 hours) / (365 days * 24 hours) = 99.9%

| Availability | Down-Time / year |
|---|---|
| 90% | > 1 month |
| 99% | ~ 4 days |
| 99.9% | ~ 8 hours |
| 99.99% | ~ 1 hour |
| 99.999% | ~ 2 minutes |
| 99.9999% | ~ 2 seconds |



## The traditional approach to stabilising a system of services is to increase the MTTF.

The **mean time to failure** is increased by

- adding redundancy in terms of servers
- adding redundancy in terms of network connectivity

**A modern approach to resilience accepts that systems crash all the time.**

The **mean time to recovery** is decreased by

- Isolation of error zones
- Loose coupling
- Redundancy
- Fallback mechanisms
- Automatisation
- Fault tolerance
- ...



## Resilience4j

**Resilience4j is a Java library that offers solutions for common resilience patterns.**

**Supported resilience patterns**

- Rate Limiter
- Retry
- Bulkhead
- Time Limiter
- Circuit Breaker



**A rate limiter is used to control the rate at which requests are processed by a system.**

- Controls request processing rate
- Prevents system overload
- Guards against traffic spikes
- Protects downstream services
- Permit-based approach

**The retry pattern is used to automatically retry a failed operation.**

- Automatically retries failed operations
- Increases reliability in case of transient failures
- Configurable maximum number of retries
- Configurable delay between retries



**A bulkhead is used to limit the number of concurrent requests to a service.**

- Isolates failures in a system
- Prevents cascading failures

**A time limiter is used to limit the execution of an operation.**

- Cancels operations if they take too long
- Prevents long-running operations from degrading system performance
- Allows for better resource management

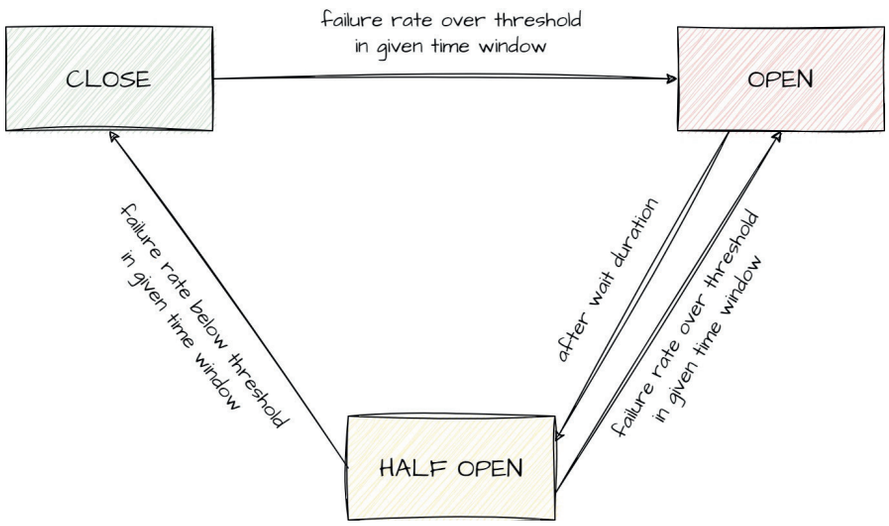


**A circuit breaker is used to detect failures and prevent further requests to a failing service.**

- Detects failures in a system
- Prevents further requests to a failing service
- Allows service time to recover

# The Circuit Breaker Resilience Pattern

**A circuit breaker monitors the success and failure rate of requests to a service.**






(1) A circuit breaker starts off in CLOSED state. This allows requests to be processed.

(2) As logn as the requests are processed successfully, it stays in CLOSED state.

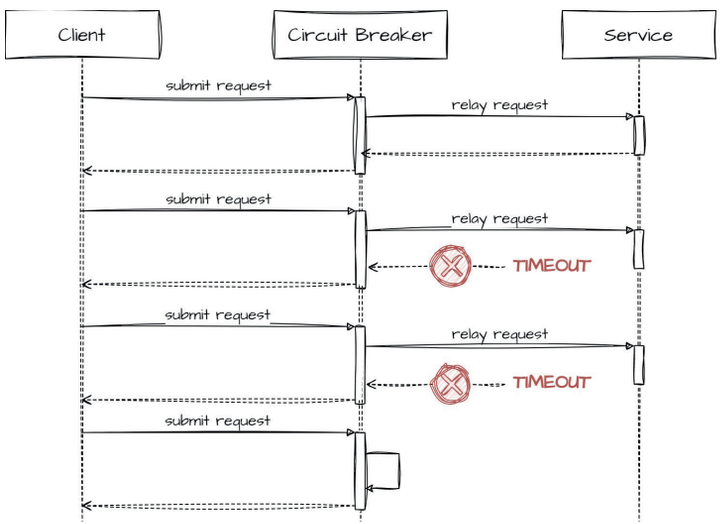
(3) A counter keeps track of failed requests.

(4) If a threshold is reached within a specified period of time, the circuit breaker will **trip**, causing it to transition from CLOSED to OPEN state. While in OPEN state, no requests are allowed to be processed. Both threshold and time window are configurable.

(5) In OPEN state, requests will **fast-fail** and raise a `CallNotPermittedException`.

(6) After a configurable period of time, the circuit breaker will enter HALF OPEN state and allow a single request to go through. This is a probe: if the request still fails, it transitions back into OPEN state, otherwise it transitions to CLOSED state, thereby allowing new requests to be processed again.

## If failures happen frequently, the circuit breaker opens and prevents further requests.






## Resilience4j offers a builder-style API for configuring a circuit breaker.

```
CircuitBreakerConfig config = CircuitBreakerConfig
  .custom()
  .failureRateThreshold(50)
  .waitDurationInOpenState(Duration.ofMillis(1000))
  .permittedNumberOfCallsInHalfOpenState(2)
  .slidingWindowSize(4)
  .build();
CircuitBreaker circuitBreaker = CircuitBreaker.of("example", config);
```

## Decorate methods that can exhibit failures using the circuit breaker.

```
Supplier<String> s = CircuitBreaker.decorateSupplier(circuitBreaker, () -> {
  System.out.println("Attempting operation");
  if (Math.random() < 0.6) {
    throw new RuntimeException("Service failure");
  }
  return "Success!";
});
```



## Resilience4j provides event listeners to monitor state transitions.

```
circuitBreaker
  .getEventPublisher()
  .onEvent(event -> System.out.println("Circuit Breaker Event: " + event));
```

Protect a service client from overloading a downstream service.

```
public String callExternalService() {
    // call external service using a circuit breaker-protected method
    try {
        return circuitBreakerProtectedSupplier.get();
    } catch (CallNotPermittedException e) {
        return "Fallback: Circuit Breaker is open";
    } catch (Exception e) {
        return "Fallback: External service call failed";
    }
}
```



# Example: Integration with Spring Cloud Gateway

## Spring Cloud offers ready-to-use compatible dependencies.

for non-reactive applications

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

for reactive applications

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>
</dependency>
```



## Use the proper `Customizer` to enable circuit breakers.

```
@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
            .circuitBreakerConfig(CircuitBreakerConfig
              .ofDefaults())
            .timeLimiterConfig(TimeLimiterConfig
               .custom()
               .timeoutDuration(Duration.ofMillis(200)).build())
            .build());
}
```

- `ReactiveResilience4JCircuitBreakerFactory` is the Spring Cloud integration piece for Resilience4j

## The Spring integration offers configurability per circuit-breaker instance.

```
resilience4j.circuitbreaker:
  instances:
    example:
      slidingWindowSize: 4
      permittedNumberOfCallsInHalfOpenState: 2
      waitDurationInOpenState: 1000
      failureRateThreshold: 50
      slowCallDurationThreshold: 2000
      slowCallRateThreshold: 100
      slidingWindowType: COUNT_BASED
      minimumNumberOfCalls: 10
```



## Use circuit breakers as part of a route configuration either programmatically ...

```
@Configuration
public class GatewayConfiguration {
  @Bean
  public RouteLocator routeLocator(RouteLocatorBuilder builder) {
    return builder
      .routes()
      .route("example_route",
              r -> r.path("/example/**")
                    .filters(f -> f.circuitBreaker(c ->
                       c.setName("example")
                        .setFallbackUri("forward:/fallback")))
                    .uri("http://localhost:8081"))
      .build();
    }
}
```

## … or via `application.yaml`

```yaml
spring:
  application:
    name: example-service
  cloud:
    gateway:
      routes:
        - id: example_route
          uri: http://localhost:8081
          predicates:
            - Path=/example/**
          filters:
            - name: CircuitBreaker
              args:
                name: example
                fallbackuri: forward:/fallback
```

## Use a fallback route in case the circuit breaker is in open state.

```
@RestController
public class FallbackController {
  @GetMapping("/fallback")
  public String fallback() {
    return "Fallback: Circuit Breaker is open";
  }
}
```



The HTTP method of the fallback endpoint must match the HTTP method of the original request. For example, if you want to implement a fallback mechanism on a route for HTTP POST requests, be sure to implement that fallback endpoint using `@PostMapping`. If the service was unable to determine a suitable fallback handler for the combination of HTTP Method + fallback endpoint, then the response `405 Method Not Allowed` will be returned. If there is no such endpoint at all, the service will respond with a `404 Not Found`. Providing fallbacks for a HTTP POST might be a tricky thing to do, though. If the circuit breaker is in HALF OPEN state, some requests may get through to probe the target service, but the fallback from the calling service still applies. Depending on what you do, this could get the user that triggered the operation a wrong sense on the outcome of the operation.

# Day 2
# Advanced Spring
## Spring Security

# Advanced Spring

## Principles of Security in Distributed Systems

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### Security in Distributed Systems

- A general issue and not caused by distributed systems
  - ... but amplified
- Resource-intensive and complex to implement
  - ... but significantly more costly and damaging when not implemented effectively.

## What's the worst that could happen?

- Temporary loss of access to business critical information
- Loss of credibility/damage to company reputation
- Temporary loss of ability to trade
- Huge cost for legal & technical consequences



"The Costliest Cyberattacks of 2022" https://securityintelligence.com/articles/13-costliest-cyberattacks-2022/ "Cost of Data Breach Report 2022": https://www.ibm.com/downloads/cas/3R8N1DZJ "Damage Control: The Cost of Security Breaches" https://media.kaspersky.com/pdf/it-risks-survey-report-cost-of-security-breaches.pdf

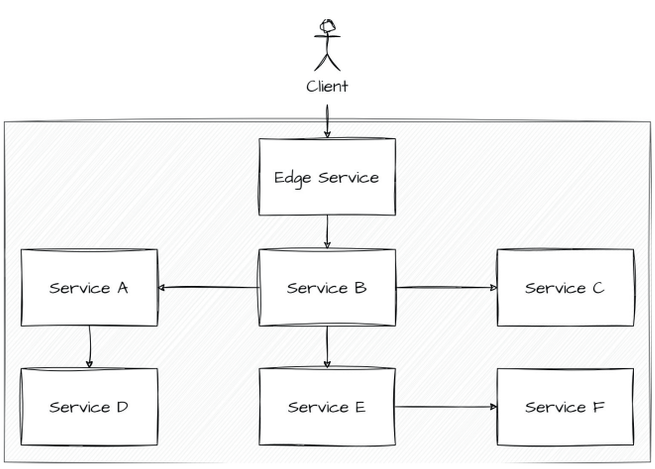
## Amplified - why?

- More systems equals more parts that can break / be broken
- *Monolith vs. Distributed* is not *Bad vs. Good* but a set of tradeoffs, we get ...
  - Loose coupling, **but** communication overhead
  - Scalability, **but** more complex deployment and monitoring
  - Smaller, optimized systems, **but** consistency issues
- Especially **communication** between systems open up additional attack vectors



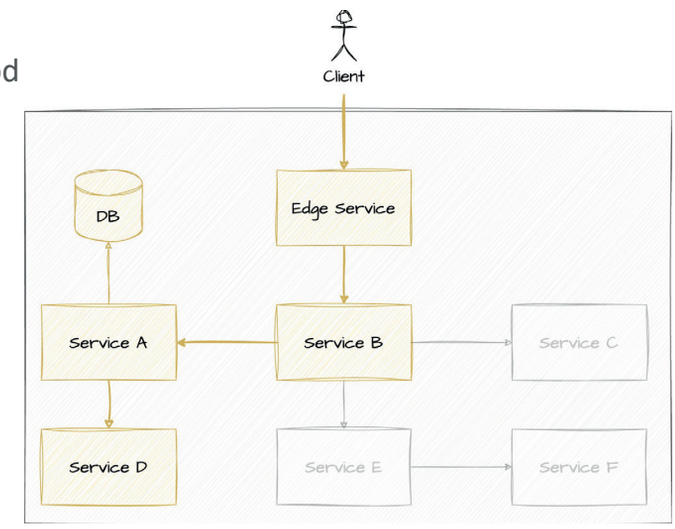
## Case study

- Edge Service example
- Clients use-case involve *Service D*
- What does that mean from a security PoV?



## Case study (cont.)

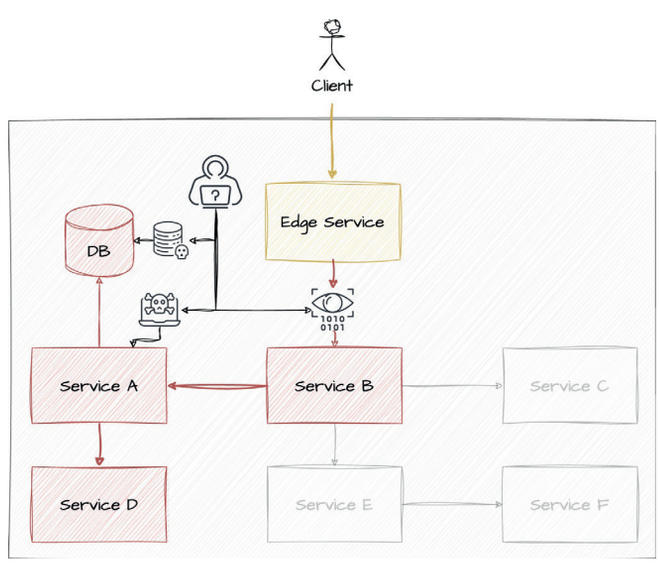
- Everything in yellow is involved
- (We added a database to *Service A* for good measure)
- **What are possible attack vectors?**
  - (within our system)






## Case study (cont.)

- **Network** (e.g. man in the middle)
  - The attacker can read information in transit
- **Storage** (e.g. weak credentials)
  - The attacker can read/manipulate information at rest
- **Service** (e.g. remote code execution)
  - The attacker can act as a service



## Case study (cont.)

**Conclusion**

- A distributed system offers potentially a lot of attack vectors
- Communication paths are not always that clear
  - proxies, network segmentation, routing, ...
- Security has to protect us from **internal** and **external** threats



# Security Requirements

## Key Requirements

What do we want to **guarantee** with security measures?

- **Confidentiality** via Encryption, Authentication
- **Integrity** via Signatures, Hashing, Consensus
- **Availability** via Load Balancing, Redundancy, Monitoring & Recovery

> *Security measures contribute to ensuring that a system runs in a predictable and reliable way.*



## Requirements - Confidentiality


- Protection from unauthorized ...
  - access
  - disclosure
  - use
- Ensures the safety of sensitive information
- The absolute basis for any form of trust in a system landscape

## Requirements - Integrity

- Assurance that data is ...
  - accurate
  - consistent
  - unaltered (except modified explicitely)
- Non-repudiation only works if we can assure integrity
- Also offers protection against data corruption due to a non-malicious source



## Requirements - Availability

- Assurance that information, systems and resources are accessible
- The basis of operational continuity
- Has the most drastic impact on user experience for **all** users

# Security Threats

## Security Threats

The security measures taken must be adequate in the context of the given security threats. Threats can be categorized in one of these categories:

- Fabrication
- Interception
- Modification
- Interruption

Every threat (except interception) is a **form of data falsification**

## Threats - Fabrication

- The creation and injection of false or malicious data
- The Intention is to deceive or disrupt normal operations
- Can be triggered over the air or via data at rest

**Example**

*An attacker has gained access to a messaging system that handles financial transactions. The attacker sends a message to transfer funds to a certain account.*



## Threats - Interception

- Unauthorized access or acquisition of (sensitive) information
- Also possible over the air or at rest

**Example**

*An attacker has managed to intercept traffic between two nodes exchanging information. These information contain credit card infos as well as user logins. This allows the attacker to login and act as the leaked user.*

## Threats - Modification

- Alteration of data **and/or** system components
- Potentially compromises the integrity of information
- Can change the behavior of a system

**Example**

> *An attacker has used a security exploit to alter the configuration of a system. The system now logs every data processed to an additional external system the attacker has full control over.*



## Threats - Interruption

- The disruption of denial of access to services / resources
- Impacts the availability and might lead to unexpected behavior

**Example**

> *An attacker has identified a critical component in a distributed landscape and floods the component with invalid requests/data. The component is rendered unable to respond to legitimate requests which could have significant consequences.*

# Security Mechanisms

## Security Mechanisms

Security mechanisms are **concrete measures** to mitigate security threats and fulfill our security requirements.

We will focus on parts of:

- Network security
- Access control and authentication
- Data security
- Monitoring, tracing and auditing

## Security Mechanisms - continued

There are more aspects to this, that are out of scope for this workshop but you should still keep in mind:

- Physical security
- Device Management
- Patch management
- Disaster recovery / incident response
- Application security
- General Awareness & training



## Security Mechanisms - Network security

Ensure the confidentiality, integrity, and availability of data and resources while in transit.

- Firewalls to filter incoming and outgoing traffic
- Intrusion Detection and Prevention Systems (IDPS) to identify and block malicious activities
- Virtual Private Networks (VPN) to secure data transmission
- Secure communication protocols (e.g., SSL/TLS, HTTPS)

## Security Mechanisms - Access control and authentication

Ensure that only authorized users can access specific resources or perform certain actions

- User authentication methods
  - passwords
  - multi-factor authentication
- Role-based access control (RBAC) to assign appropriate permissions
- Single sign-on (SSO) solutions for simplified and centralized authentication



## Security Mechanisms - Data Security

Protect sensitive information from unauthorized access, disclosure, alteration, or destruction, both when stored and during transmission, ensuring the confidentiality and integrity of the data.

- Encryption for data at rest and in transit
- Data masking and anonymization
- Data backup and recovery solutions

## Security Mechanisms - Monitoring and tracing

Continuously observing, logging, and analyzing system activities to detect and respond to potential security threats and maintain the overall health and performance of IT systems.

- Systems to collect and analyze events & logs
- Continuous monitoring and logging of system activities
- Exposure of endpoints that allow the collection of metrics

# Advanced Spring

## Spring Security

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### Introduction

- **Authentication** and **Authorization** framework for Java applications
- Highly customizable security framework
  - Integrated with Spring ecosystem
- Most recent version is 6.x

> *First class support for securing both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.*

# Authentication

## Authentication

- Verifying user/system identity
- Supports various authentication mechanisms:
  - Username/password
  - OAuth2/OIDC
  - SAML
  - Custom
  - ...

## Authentication (cont.)

### Main components

- Authentication Manager
- Authentication Provider
- Password Encoder
- User Details Service



## Authentication Manager

- Central authority for authentication
- Single method: `authenticate()`

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication) throws AuthenticationException;
}
```

- Return `Authentication` (with `authenticated=true`) if the input represents a valid principal
- Throw an `AuthenticationException` if the input represents an invalid principal
- Return `null` if it cannot decide
- Default implementation is the `ProviderManager` that delegates to a chain of `AuthenticationProvider`

## Authentication Provider

- An `AuthenticationProvider` has an extra method to allow the caller to query whether it supports a given `Authentication` type.
- Does **not** extend the `AuthenticationManager` !

```
public interface AuthenticationProvider {

Authentication authenticate(Authentication authentication) throws AuthenticationException;

boolean supports(Class<?> authentication);

}
```



## Authentication Provider (cont.)

- Responsible for one specific way of authentication
- Decision is made via the `supports` method if the provider is applicable

```
class OidcAuthenticationRequestChecker implements AuthenticationProvider {

(...)

    public boolean supports(Class<?> authentication) {
        return OAuth2LoginAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

## Password Encoder

Interface to perform a one-way transformation of a password to let the password be stored securely

- Default is the `DelegatingPasswordEncoder` that understands different password hashes
- The format might look familiar to you:

```
{id}encodedPassword
{bcrypt}`$2a$`10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
{noop}password
{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc
{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbc0
```



## User Detail Service

- Interface for retrieving user details
- Can use various data stores (e.g., database, LDAP, in-memory)
- `UserDetails` is an interface that describes the minimal set of user information

```
public interface UserDetailsService {

    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}
```

# Authorization

## Authorization

- Determining and validating permissions for resources/actions
- Main components
  - Security Interceptor
  - Authorization Manager (formerly Access Decision Manager)
  - Granted Authority / Role

## Security Interceptor

- Intercepts incoming requests
- Checks user permissions
- Commonly used examples: `FilterSecurityInterceptor`, `MethodSecurityInterceptor`



## Authorization Manager

- Since Spring Security 6 the `AccessDecisionManager` is deprecated
- Makes final access control decision
- Takes an `Authentication` object and decides for a given resource if the permissions are sufficient
- Can be used with a different set of strategies to make the final decision:
  - `AuthorizationManagers.anyOf` - at least one of the managers grant access
  - `AuthorizationManagers.allOf` - all of the managers grant access
  - Consensus based decision is now longer provided out of the box

### GrantedAuthority vs Role

A `GrantedAuthority` and `Role` are technically the same construct with different semantics

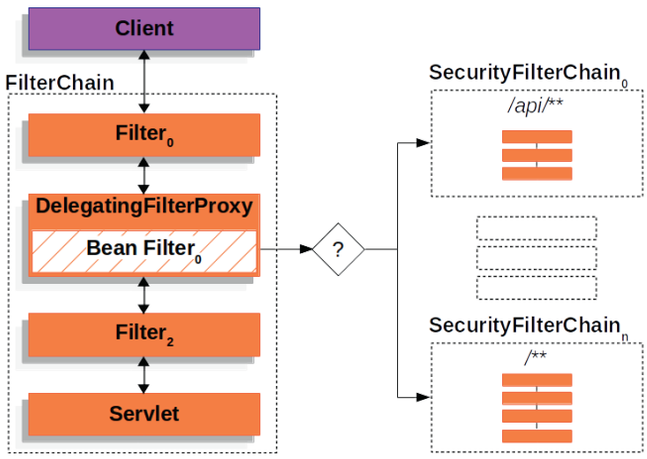
- A `GrantedAuthority` is an individual permission with an **arbitrary name**
  - Is checked via `hasAuthority` - e.g. `@PreAuthorize("hasAuthority('WRITE_ENTRY')")`
- A `Role` is a `GrantedAuthority` container with the prefix `ROLE_` (per default)
  - Is checked via `hasRole` - e.g. `@PreAuthorize("hasRole('ROLE_EDITOR')")`



## Securing Web Resources

## Architecture

- Spring Security is executed as part of the `FilterChain`
- `SecurityFilter` are a chain again
- There can be 1...n `SecurityFilterChain` - one per context
- Ordering matters - **a lot!**

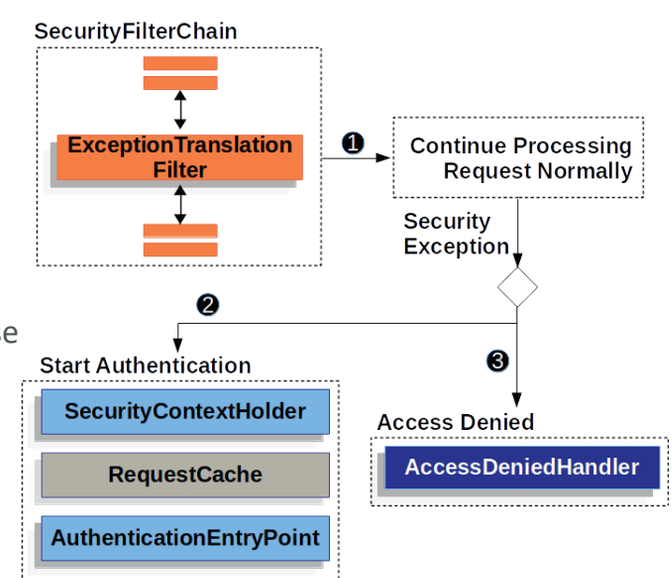





## Exception Handling

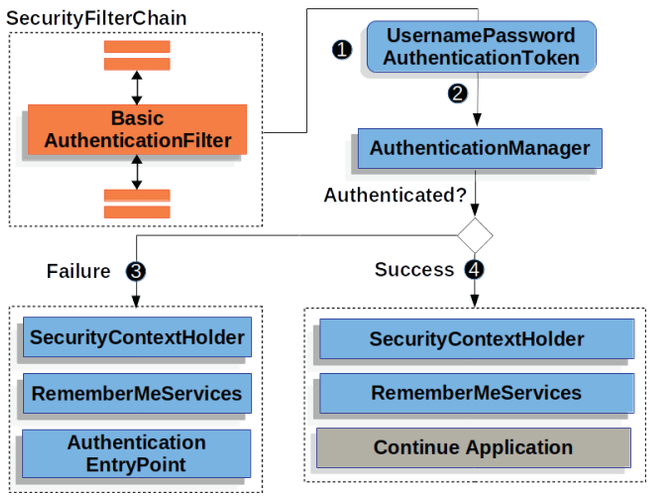
Two types of exception: `AuthenticationException` and `AccessDeniedException`

- `AuthenticationException` - start a new authentication if there is a `AuthenticationEntryPoint`, otherwise `401 - Unauthorized` for HTTP
- `AccessDeniedException` - terminate the request `403 - Forbidden` for HTTP



## Basic Auth Example

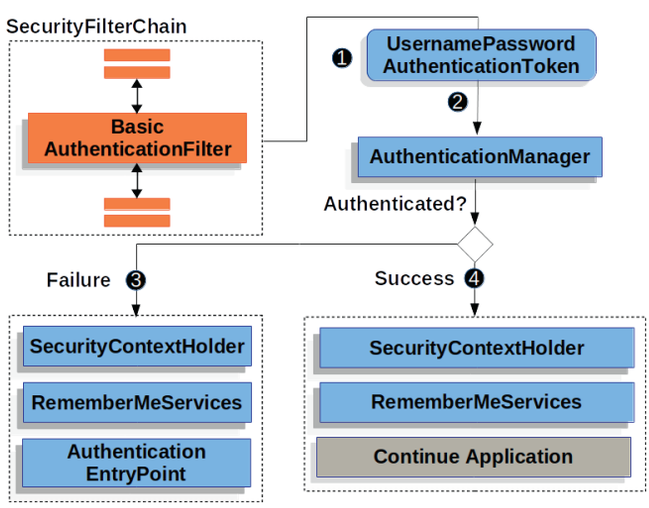
- `BasicAuthenticationFilter` is triggered as part of the `SecurityFilterChain`
- UPAToken is a specialized version of `AbstractAuthenticationToken`
- The appropriate `AuthenticationManager` validates the token against a `UserDetailsService`
- The result is saved into the `SecurityContextHolder` (thread based)






## Basic Auth Example (cont.)

- The `SecurityFilterChain` is configured with matchers
- This is the basis for the decision which `Filter` should be run in which Context in which order



## Basic Auth Example (cont.)

- `@EnableWebSecurity` on a `@Configuration` class to configure `SecurityFilterChain`
- `@EnableMethodSecurity` to use method-level security annotations
- Matcher to enable Basic Auth for all matching URIs

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain
      (HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(requests -> requests
          .requestMatchers("/**")
          .authenticated())
        .httpBasic(Customizer.withDefaults());

    return http.build();
  }
}
```



## Basic Auth Example (cont.)

- `@PreAuthorize` to check for a specific role
- `Authentication` object will be injected from the current SecurityContext
  - `Principal` is also possible

```
@RequestMapping("/hello")
@RestController
public class HelloWorldController {

  @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
  @PreAuthorize("hasRole('USER')")
  UserPermissions helloUser(Authentication authentication) {
    final var authorities = authentication.getAuthorities().stream()
        .map(GrantedAuthority::getAuthority)
        .toList();
    return new UserPermissions(authentication.getName(), authorities);
  }
}
```

# Lab

## Lab

It's time to use (*some*) of that!

- Open the Spring Security Repository in your IDE
- Let's take a look at the repository and README.md

# Advanced Spring

## Distributed Tracing with Spring

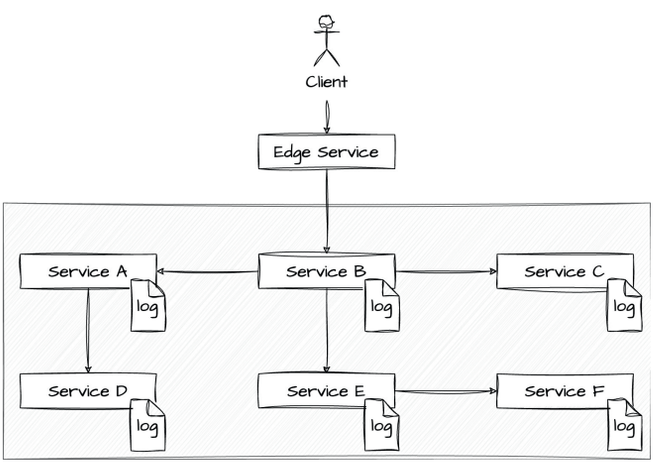
**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### Some difficult questions

- Which services are involved in a use-case?
- Where are our bottlenecks?
- Why do we see so many 4xx / 5xx in our logs?
- Are we hitting our SLOs/NFRs?



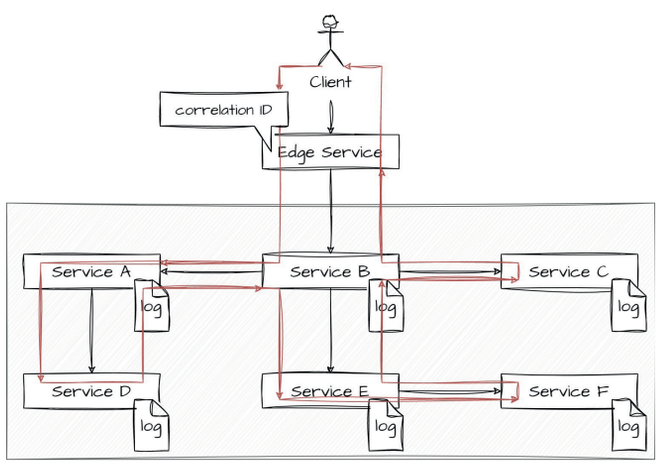
## Distributed Tracing

*Distributed tracing provides end-to-end visibility into the performance and behavior of requests across interconnected services in a distributed system, enabling faster troubleshooting and optimization of the application.*



## The solution

- A unique ID that is passed from system to system
- All systems log into a central component
- Traces with the same ID are combined into a single trail



**Key Points**

- (Enhanced) Observability
  - Comprehensive view of interaction between services
- Faster root cause analysis
  - Especially for Bottlenecks / Latency issues
- Service Dependency Visibility
  - Get a per use-case view of all involved services
- Anomaly Detection
  - Basis for monitoring, especially for non-functional requirements



## Components

## Conceptual Components

- **Instrumentation**
  - Capture & record information about performance characteristics
- **Propagation**
  - Provide the necessary information to correlate further operations
  - Pass HTTP-Headers for HTTP Requests
  - Log the span- / trace-id with the service logs for later analysis
- **Collection**
  - Aggregate the collected data for a context



## Conceptual Components (cont.)

- **Storage & Processing**
  - Persist data received
  - Correlate the related datasets
- **Visualization & Analysis**
  - User Interface for the data
  - Capability to search & filter
- **Alerting & Integration**
  - Automated analysis based on the collected data

## Libraries and Systems

- **Brave**: Tracer Library for Instrumentation (developed and maintained by Zipkin)
- **Spring Cloud Sleuth**: Layer on top of Brave for smooth Spring integration
  - *No longer available for Spring Cloud 2022.x*
- **Micrometer Observation**: Vendor/technology agnostic metrics collector for Java
  - Can be used with Brave & Zipkin or other technologies
  - *The new default for Spring Cloud 2022.x*
- **Zipkin**: Distributed Tracing System for collection & visualization



## Data Model

- **Trace**: Unique object that contains 1..n spans - builds a latency tree
- **Span**: Single host-view of an operation. Most important fields are:
  - `traceId`: the root trace ID
  - `parentId`: the ID of the parent span (null if root)
  - `name`: the name of the span
- **Tag**: A marker for a span - can be used in a query
- **Annotation**: A specific point in time in a span with an attached string value

# Spring Cloud Sleuth

## Spring Cloud Sleuth

*Spring Cloud Sleuth will not work with Spring Boot 3.x onward. The last major version of Spring Boot that Sleuth will support is 2.x.*

We'll only take a quick look at Sleuth for the sake of legacy knowledge

## Integration

- Easy integration via Spring Cloud dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

- This includes the auto configuration part that ...
  - exposes Java beans for customizing data collection & presentation
  - decorates existing beans (e.g. `RestTemplate` for propagation)



## Configuration

- The default configuration should be changed - for our example we used this config

```
spring.application.name: Client Application

spring:
    zipkin:
        base-url: ${ZIPKIN_API_ENDPOINT:http://127.0.0.1:9411/}
    sleuth:
        sampler:
            probability: 1.0
```

- Application name will be reported with the spans
- The Base-URL is where the traces will be sent to
- The probability default is `0.1` - so only 10% of traces would be gathered
  - Crucial overload protection for high load systems

## Creating Traces

- Spans will be created and reported implictly for supported operations
  - `@Scheduled`, `@RestController`, ...
  - The current span can also be modified
- Can also be done explicitly
  - Via annotations
  - Programatically



## Creating Traces via Annotations

- `@NewSpan("name")` tells Sleuth to open a new span with the given name
  - Will be the child of the already existing span - if there is one
- `@ContinueSpan` can be used to annotate the current span
- `@SpanTag` can be used on parameters to add them as tags

```
@NewSpan("user-lookup")
public User lookupUserByName(@SpanTag(key = "username") final String username) {
  (...)
}
```

## Creating Traces programmatically

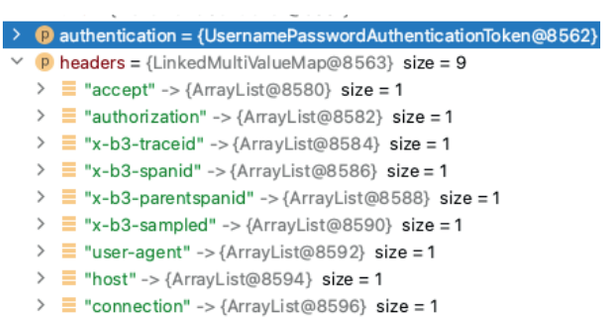
- `@NewSpan("name")` tells Sleuth to open a new span with the given name
  - Will be the child of the already existing span - if there is one
- `@ContinueSpan` can be used to annotate the current span
- `@SpanTag` can be used on parameters to add them as tags

```
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
  newSpan = this.tracer.nextSpan().name("calculateCommission");
  newSpan.tag("username", username)
} finally {
  if (newSpan != null) {
    newSpan.finish();
  }
}
```



## Propagation

- Sleuth will add the tracing context to outgoing requests
  - will work out of the box for directly supported mechanisms (e.g. `RestTemplate`)
- For HTTP requests HTTP-Headers (`x-b3-`) will be added
- Instrumentation library on the receiving side will continue the context



# Micrometer Observation

## Micrometer Observation

- Micrometer is a very powerful library for collecting metrics
  - You will hear more about this tomorrow!
- The Micrometer Observation API is vendor neutral
  - Modular approach allows a *do once, use many times* approach.
  - Has a bridge to Brave and therefore Zipkin

## Integration

- Dependencies are not from Spring
- Requires out-of-band dependency management
- Some integration with `spring-boot-starter-actuator` for autowiring beans

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```



## Integration (cont.)

- Has a dedicated dependency for test support
  - Very useful - we will take a look at this tomorrow

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation-test</artifactId>
    <scope>test</scope>
</dependency>
```

## Architecture

- `Observation` is a wrapper class for an observable operation
- `Observations` follow a life cycle
  - `start`: The operation has begun execution
  - `scope started`: A scope within the operation has been opened
  - `scope ended`: A scope was closed
  - `event`: An event has happened while observing (e.g. annotations in Sleuth)
  - `error`: The operation threw an error
  - `stop`: The operation has ended



## Architecture

- `Observations` are handled by `ObservationHandlers`
  - The signature is `ObservationHandler<T extends Observation.Context>`
  - The interface provides handling methods for all lifecycle events
- Micrometer provides an `ObservationRegistry` to register `ObservationHandler`
  - `ObservationHandler` are called when they support a specific `Observation.Context`
- The `Observation.Context` is a mutable data holder to share data between states
  - The context will be propagated between threads!

## Configuration

```
spring.application.name: Micrometer Tracing
logging.pattern.level: "%5p [`${spring.zipkin.service.name:$`{spring.application.name:}},%X{traceId:-},%X{spanId:-}

management:
    tracing:
        propagation:
            type: b3
        sampling:
            probability: 1.0
    zipkin:
        tracing:
            endpoint: ${ZIPKIN_API_ENDPOINT:http://127.0.0.1:9411/api/v2/spans}
```

- Probability and endpoint are similar (plus path in URL) to Sleuth
- We need to reconfigure the logging pattern manually to log trace- and span-IDs
- Propagation type `b3` for interop with Sleuth (default is `w3c`)



## Configuration (cont.)

- To use the `@Observed` annotation we need to register the Bean for AOP
- The `ObservationRegistry` is the registry responsible for observation state management

```
@Bean
public ObservedAspect observedAspect(ObservationRegistry observationRegistry) {
  return new ObservedAspect(observationRegistry);
}
```

## Creating Observations via Annotations

- `@Observed` tells Micrometer that the method should be observed

```
@Observed(
  name = "time.reporting",
  contextualName = "time-report",
  lowCardinalityKeyValues = {"class.name", "TimeReportingTask"}
)
public void reportTimeTask() {
  (...)
}
```



## Creating Observations programatically

- `Observation` offers several static factory methods to create a new `Observation`
- `observe` will
  - start/stop the `Observation` (if not already done)
  - Open/close a new scope
  - Catch all errors

```
void observeWork(@Autowired ObservationRegistry registry) {
  final var context = new Observation.Context().put(String.class, "information");
  // The context is optional
  Observation
    .createNotStarted("operation.name", () -> context, registry)
    .observe(this::doWork);
}
```

## Manipulate Observations

- The `ObservationRegistry` is the global holder for all `Observations` (comparable to the `Tracer` in Sleuth)
- Fetch current observation via `registry.getCurrentObservation()`
  - Will be `null` if we're not being observed

```
void doWork(@Autowired ObservationRegistry registry) {
  final var observation = registry.getCurrentObservation();
  (...)
}
```



## Observations Bridge

- An `Observation` conceptually has nothing to do with a trace
- `micrometer-tracing` provides a set of `ObservationHandler` to handle `Observations`
  - Transitive dependency of `micrometer-tracing-bridge-brave`
- `micrometer-tracing-bridge-brave` is responsible for
  - converting the traces to Brave
  - Reporting to Zipkin via Brave

# Zipkin

## Zipkin

*Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.*

## Zipkin (cont.)

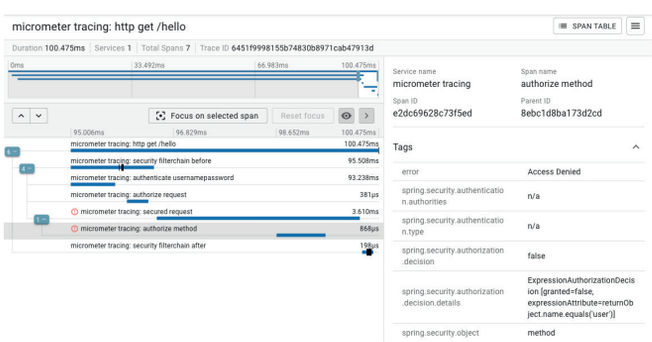
- Zipkin provides
  - a HTTP Facade to receive (and query!) spans
  - an UI for visualization & querying
- Is usually run as a standalone application
  - but can be integrated in a service as well
- Spans can be collected via HTTP, Kafka or RabbitMQ
- Storage options are in-memory, MySQL, Cassandra or Elasticsearch
  - There are a lot of community supported plugins e.g. for SQS, Kinesis



## Zipkin (cont.)

**Lab assignments will focus on**

- visualization & querying of data
- tracing HTTP Requests



# Lab

## Lab

It's time to get started!

Let's take a look at the repository and README.md

# Day 3
# Advanced Spring
## Migration, Testing, Metrics

# Advanced Spring

## Spring Boot Migration Guide

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

## Before migrating to Spring Boot 3

**Satisfy prerequisites for the migration.**

- JDK 17 is the minimum JDK for Spring Boot 3.0
- Upgrade to the latest 2.7.x version first

## Review existing dependencies.

- Spring offers comprehensive overviews of its dependency hull
- Compare 2.7.x with 3.0.x to identify where you're affected
- Watch out for compatible Spring Cloud dependencies
  - 2022.0.x (Kilburn) is compatible with 3.0.x
- Spring Boot 3.0 uses Spring Security 6.0



The dependency hull for 2.7.x is available at https://docs.spring.io/spring-boot/docs/2.7.x/reference/html/dependency-versions.html. The dependency hull for 3.0.x is available at https://docs.spring.io/spring-boot/docs/3.0.x/reference/html/dependency-versions.html. Spring Cloud dependencies are not managed by Spring Boot. Update any Spring Cloud dependencies to the 2022.0.x line if you haven't done so already. Upgrade your 2.7.x application to use Spring Security 5.8. Upgrading to Spring Security 5.8 first significantly lowers the bar of the whole migration.

## Upgrade to Spring Security 5.8 before migrating to Spring Boot 3.0.

- Easy to upgrade from Spring Security 5.8 to Spring Security 6.0
- Necessary migrations can be done with 5.8 against your stable 2.7.x application



## Update password encoding in case you use PBKDF2, SCrypt or Argon2.

- Standards changed for PBKDF2, SCrypt, and Argon2
- Replace deprecated constructor usage by static factory

```
@Bean
PasswordEncoder encoder() {
  return Pbkdf2PasswordEncoder.defaultsForSpringSecurity_v5_5();
}
```

## Use `DelegatingPasswordEncoder` to future-proof your password encoding strategy.

- Security recommendations change and you need to adapt
- `DelegatingPasswordEncoder` supports multiple `PasswordEncoder` implementations
- Selects algorithm at runtime, prefixes password hashes with strategy
- Database can contain password hashes using different algorithms

```
@Bean
PasswordEncoder encoder() {
  String prefix = "pbkdf2@5.8";
  PasswordEncoder current = // ... see the previous slide
  PasswordEncoder upgraded = Pbkdf2PasswordEncoder.defaultsForSpringSecurity_v5_8();
  DelegatingPasswordEncoder delegating = new DelegatingPasswordEncoder(prefix, Map.of(prefix, upgraded));
  delegating.setDefaultPasswordEncoderForMatches(current);
  return delegating;
}
```



## Stop using `Encryptors.queryableText`!

- `Encryptors.queryableText(CharSequence, CharSequence)` is unsafe
  - same input produces same input
  - removed in Spring Security 6.0
- To upgrade either re-encrypt or store decrypted

```
TextEncryptor deprecated = Encryptors.queryableText(password, salt);
BytesEncryptor aes = new AesBytesEncryptor(password, salt, KeyGenerators.secureRandom(12), CipherAlgorithm.GCM);
TextEncryptor supported = new HexEncodingTextEncryptor(aes);
for (MyEntry entry : entries) {
    String value = deprecated.decrypt(entry.getEncryptedValue());
    entry.setEncryptedValue(supported.encrypt(value));
    entryService.save(entry)
}
```

## Reference Jakarta EE dependencies in your roles-and-rights implementation.

### With Spring Boot 2.x

- `javax.persistence.Entity`

```
import javax.persistence.Entity;
import javax.persistence.Entity;

@Entity
@Table(name="users")
public class User implements Serializable {
  /* implementation goes here /*
}
```

### With Spring Boot 3.x

- `jakarta.persistence.Entity`

```
import jakarta.persistence.Entity;
import jakarta.persistence.Table;

@Entity
@Table(name="users")
public class User implements Serializable {
 /* implementation goes here /*
}
```



## Spring Security 6.0 removes `WebSecurityConfigurerAdapter` and renames methods.

### With Spring Boot 2.x

- `authorizeRequests()`
- `antMatchers()`
- `regexMatchers()`

```
@Configuration @EnableWebSecurity
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http)
      throws Exception {
    http.authorizeRequests()
      .antMatchers("/home", "/register", "/saveUser")
      .permitAll();
  }
}
```

### With Spring Boot 3.x

- `authorizeHttpRequests()`
- `requestMatchers()`
- `regexRequestMatchers()`

```
@Configuration @EnableWebSecurity
public class WebSecurityConfig {
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http)
      throws Exception {
    return http.authorizeHttpRequests()
      .requestMatchers("/home", "/register", "/saveUser")
      .permitAll()
      .build();
  }
}
```

# Migrating to Spring Boot 3

## Upgrade to the latest maintenance release of Spring Boot 3.0.

- Increase the dependency version
- Let Maven / Gradle do its job
- Fix compilation errors and deprecated elements

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.0</version>
  <relativePath></relativePath>
</parent>
```



It's time to upgrade to Spring Boot 3.0. Always aim to use the latest maintenace release; in this case, we'll use version 3.0.0. To upgrade, open your project's `pom.xml` file and update the Spring Boot version as shown. After saving the updated `pom.xml`, Maven will download the new dependencies. During this process, Maven will handle most of the upgrade work, ensuring that all required dependencies are compatible with Spring Boot 3.0.0.

## Migrate configuration properties.

- Spring Boot 3.x renamed and removed a couple of properties
- Spring Boot provides a migration toolkit

### Steps

1. Add the dependency
2. Build and package
3. Launch JAR
4. Migrator prints a migration report

### Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-properties-migrator</artifactId>
  <scope>runtime</scope>
</dependency>
```

### Renamed

- `spring.redis` to `spring.data.redis`
- `spring.data.cassandra` to `spring.cassandra`
- …

### Removed

- `spring.jpa.hibernate.use-new-id-generator`

## Spring Boot Properties Migrator prints a comprehensive migration report.

**Renamed properties**

```
The use of configuration keys that have been renamed was found in the environment:

Property source 'Config resource 'class path resource [application.yml]' via location
'optional:classpath:/'':
    Key: spring.resources.cache.period
        Line: 2
        Replacement: spring.web.resources.cache.period
```

**Removed properties**

```
The use of configuration keys that are no longer supported was found in the environment:

Property source 'Config resource 'class path resource [application.yml]' via location 'optional:classpath:/'':
    Key: spring.resources.chain.html-application-cache
        Line: 4
        Reason: none
```

**Update package references for Java EE related dependencies.**

- Java EE changed to Jakarta EE
- Spring Boot reflects that change with 3.0
- Spring Boot uses Jakarta EE 10 compatible dependencies

| Prior to 3.x | With 3.x |
|---|---|
| `javax.persistence.*` | `jakarta.persistence.*` |
| `javax.validation.*` | `jakarta.validation.*` |
| `javax.servlet.*` | `jakarta.servlet.*` |
| `javax.transaction.*` | `jakarta.transaction.*` |



If you don't utilize Spring Boot's starter dependencies but your own dependency management, check if your dependency hull reflects the necessary changes as well. Please note that changing `javax` to `jakarta` is not a general guideline! For instance, packages such as `javax.sql.*` and `javax.crypto.*` will not be affected by the transition from Java EE to Jakarta EE, since these packages are part of the JDK itself.

# After migrating to Spring Boot 3

**Spring MVC and WebFlux changed their behavior when matching URLs.**

- Spring 6.0 deprecates the trailing slash matching configuration option
- Previously this controller would have matched
  - `GET /employee/register`
  - `GET /employee/register/`

```
@RestController
public class EmployeeController {
  @GetMapping("/employee/register")
  public String showRegistrationPage() {
    return "registration";
  }
}
```

## Spring MVC and WebFlux changed their behavior when matching URLs. (cont.)

- Don't insert trailing slashes while matching URLs
- Update client components as appropriate
- Use the following configuration in the meantime

```
@Configuration
public class WebConfiguration implements WebMvcConfigurer {
  @Override
  public void configurePathMatch(PathMatchConfigurer configurer) {
    configurer.setUseTrailingSlashMatch(true);
  }
}
```



## The default format for Logback and Log4j2 has changed.

- New format is in line with ISO 8601
- Format: `yyyy-MM-dd'T'HH:mm:ss.SSSXXX`
  - note the T character rather than a space
  - adds timezone offset
- Previous format: `yyyy-MM-dd HH:mm:ss.SSS`
- Override / restore
  - `logging.pattern.dateformat` (property)
  - `LOG_DATEFORMAT_PATTERN` (env)

## @ConstructorBinding is no longer required at the type level.

- Indicates that properties should be mapped using the constructor
- No longer necessary on a class annotated with `@ConfigurationProperties`

```
@ConfigurationProperties(prefix = "my-properties")
// @ConstructorBinding
public class PropertiesConfig {
  private String name;
  private String description;

  public PropertiesConfig(String name, String description) {
    this.name = name;
    this.description = description;
  }
}
```



## Actuator endpoints mask the values for all keys by default.

- Spring Boot Actuator previously masked only values for *sensitive* keys
  - affects endpoints `/env` and `/configprops`
- Spring Boot 3.x opts to be more secure
  - all values masked by default
  - override `management.endpoint.{env|configprops}.show-values`
  - admissible values: `NEVER`, `ALWAYS`, `WHEN_AUTHORIZED`

**Spring Boot 3.0 removes support for registering auto-configs in `spring.factories`.**

- Spring Boot 2.7 introduced a new way to register auto-configurations
  - `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguratic`
  - backwards-compatible with `spring.factories`
- Using `spring.factories` is no longer possible
- Use the `imports` file instead



**The support for image-based banners has been removed.**

- Apparently banners are a popular feature
- It *was* possible to use a `banner.{gif|jpg|png}`
- Restricted to textual content with Spring Boot 3.0 (`banner.txt`)

# Advanced Spring

## Testing with Spring - Best Practices

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### The problem

- We want coding to be as comfortable as possible
- That means
  - using many libraries/frameworks
  - using the Spring application context
  - focussing on business logic and less on boilerplate

But if a lot happens out of our control - how do we test that efficiently?

## The problem (cont.)

The naive approach is to use `@SpringBootTest` for everything.

- What about external dependencies? (databases)
- Things we **don't** want to run (e.g. `@Scheduled` tasks while testing a REST facade)
- Systems that are not even available locally (e.g. message brokers)



## The problem (cont.)

Even if all of this is a non-issue in a specific case

**Starting the whole container is incredibly slow!**

# Principles and Best Practices

## F.I.R.S.T. Principles

What **SOLID** is to software architecture, **FIRST** is for software testing: a good rule of thumb

- **Fast**: Running the tests should be fast enough to run all of them. Always.
- **Isolated**: No interdependencies between tests or other external factors
- **Repeatable**: Testruns must be deterministic, no matter the environment.
- **Self-validating**: Tests should indicate directly if they're successful or not
- **Thorough**: Try to cover everything
  - All happy paths & edge cases
  - All combinations of possible input (e.g. `null`)

## Spring Boot Testing

- Use provided slices whenever possible - e.g. `@DataJpaTest` or `@WebMvcTest`
  - Slices only load the minimal amount of necessary Beans
- Don't mess with the test context unless absolutely necessary
  - This will either lead to brittle tests or cache misses
  - Use `@DirtiesContext` with great care
- Use mocking (with Mockito or other libraries)
  - Mocking POJOs is an antipattern



## Spring Boot Testing (cont.)

- Don't reinvent the wheel
  - Use test libraries from the vendors where available, e.g. `micrometer-observation-test`
  - Make full use of all the assertions available in JUnit or use a powerful library like AssertJ
- Avoid code redundancy
  - Write helper classes/methods for shared issues, e.g. object creation
  - Use the `@Before` / `@After` and other annotations

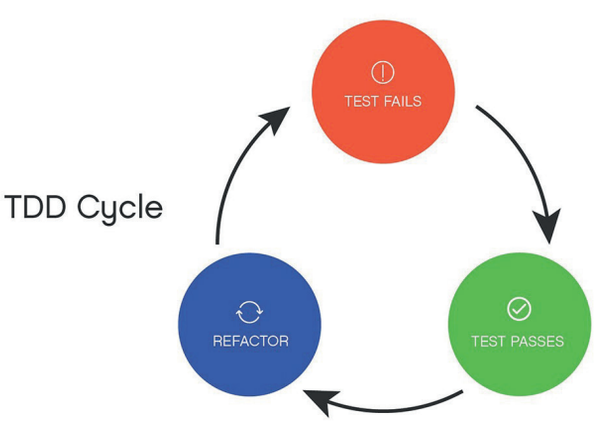
## Spring Boot Testing (cont.)

- Write short, concise tests
  - Your unit tests ideally tests exactly one thing
  - Your unit tests should not need tests to be understood
- Use unit tests vs. integration tests whenever possible
  - Spring sometimes blurs the line
  - Prefer to use no context if possible



## Spring Boot Testing (cont.)

- Use insights from your test cases to refactor your code
  - Code that is hard to test is a code smellsmell
  - TDD avoids that issue but is not always applicable (e.g. legacy applications)
- Strong tests should still be the foundation of every refactoring



## Examples

Talking about testing is cheap!

**Let's take a look at some common mistakes and concrete examples in the IDE.**

# Advanced Spring

## Monitoring Business-Related Metrics

**Boris Fresow, Markus Günther**

Adesso eduCamp 2023

Mastichari, Kos

### Why do we care for metrics anyways?

*Collecting and analyzing metrics is essential for an organization to get a thorough understanding of a product's performance.* ***Technical metrics*** *ensure the* ***soundness of the software****, while* ***business metrics*** *help to align the software's performance with* ***business goals*** *of the organization.*

# Business metrics vs. technical metrics

**Business metrics have a different *focus* than technical metrics.**

**Business metrics**

- Performance of the product
- Customer behavior
- Financial performance
- Critical aspects of the business

**Technical metrics**

- System performance
- Stability
- Reliability

# Advanced Spring – Business-Related Metrics

**Business metrics serve a different *purpose* than technical metrics.**

**Business metrics**

- Evaluate the success of a product
- Guides informed decisions
- Helps to achieve business goals

**Technical metrics**

- Develop high quality systems
- Provide a solid foundation for the product

## Examples

**Business metrics**

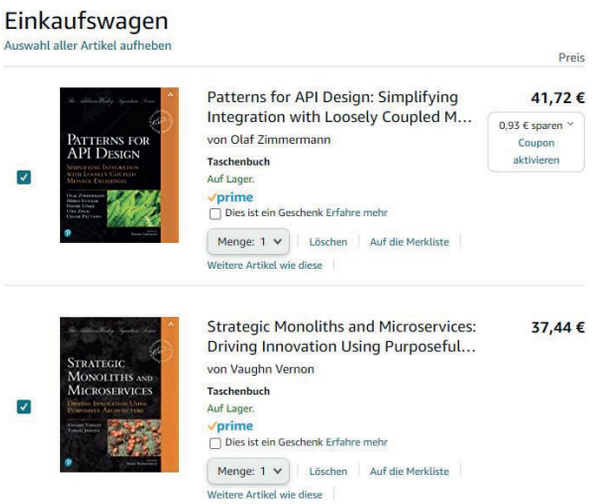
- User Sign-ups
- Conversion Rate
- Revenue
- Churn Rate
- Click Rank

**Technical metrics**

- Response Time
- Service Time
- Error Rates
- Resource Utilization
- Availability
- Throughput
- Code Quality

(1) User Sign-ups refers to the number of new users that registered for your product. This metric is a key indicator for growth. (2) Conversion rate refers to the percentage of users that completed a desired action, e.g. closing an order in an E-commerce application or registering for a newsletter. It is a key indicator for the effectiveness of your marketing strategy and user experience. (3) Revenue is the total income generated by your product, preferably over a specific period of time. (4) Churn Rate refers to the percentage of users who discontinue using the product. Low churn rate is an indicator for customer happiness, a high churn rate signals customer dissatisfaction.

## What's in a metric like conversion rate?





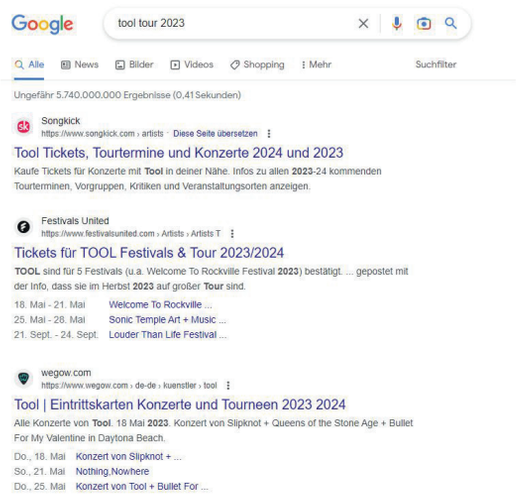

## What's in a metric like conversion rate? (cont.)

### Building blocks

- counter for completed orders
- counter for open orders / sessions

$$\text{conversion rate} = \frac{\text{number of completed orders}}{\text{number of open cart sessions}}$$

## What's in a metric like click rank?






## What's in a metric like click rank? (cont.)

### Building blocks

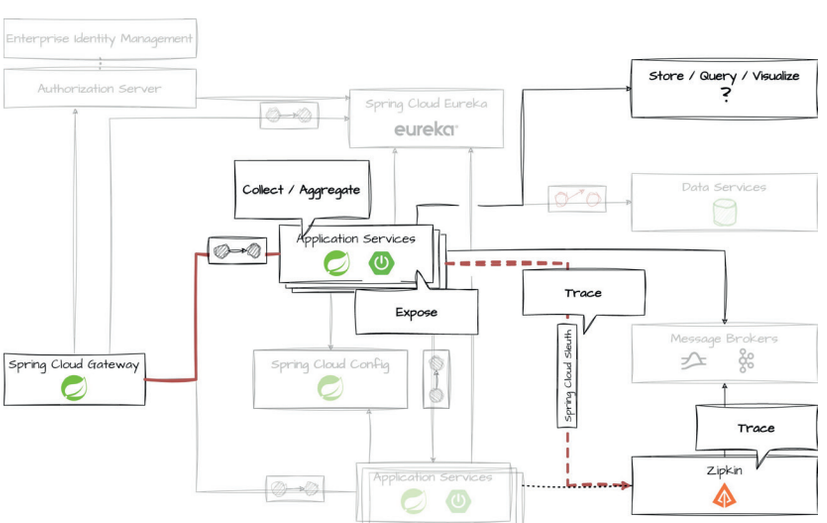
- rank in search result list
- the user query
- additional search context
- information on target document
- ...

### Represent as structured content

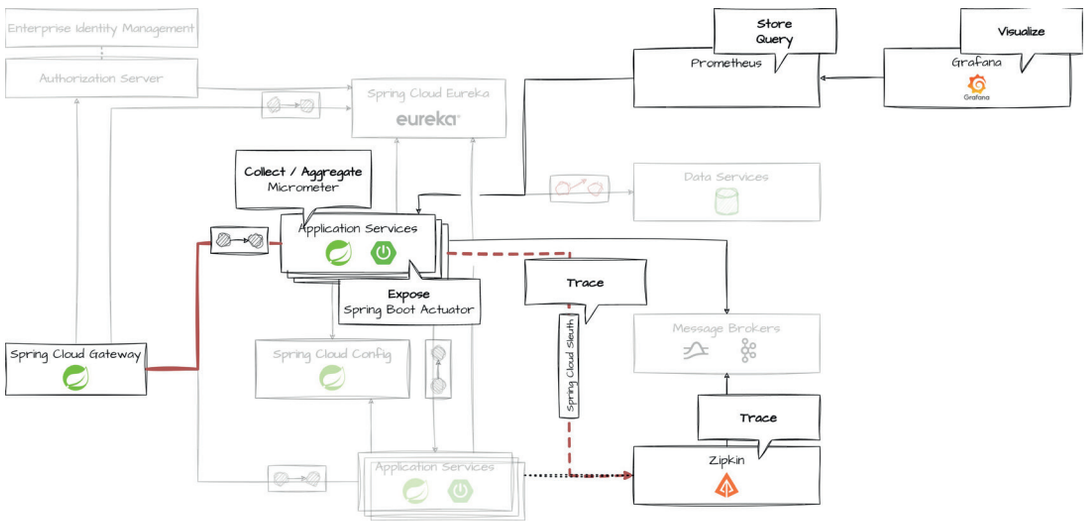
```
{
  "query": "tool tour 2023",
  "rank": 3,
  "title: "Tool | Eintrittskarten Konzerte und [...]",
  "document": "https://www.wegow.com/de-de/kuenstler/tool",
  "facets:": [],
  "userSession": "7d7b9c28-e52c-11ed-9365-00155d21c55c",
  [...]
}
```

# Monitoring Tools and Techniques

**The acquisition of metrics revolves around a couple of concepts.**



**The Spring ecosystem offers libraries and integrations to support the acquistion of metrics.**






# Collect and Aggregate

**using Micrometer**

## Micrometer is the SLF4J for metrics.

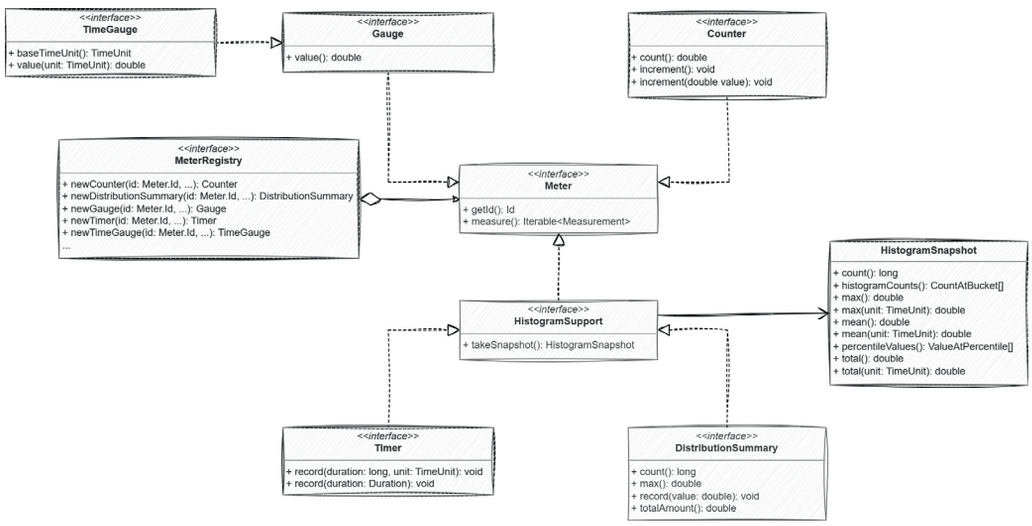
- Easy-to-use
- Vendor-neutral
- Offers a variety of useful instruments
- Integrates with a variety of monitoring tools
  - e.g. Prometheus



## Micrometer provides a rich set of instruments for capturing metrics.

- Counter
- Gauge
- Timer
- Distributions
- ...

## A `Meter` is the base abstraction for all specific instruments.






The `MeterRegistry` is the central component responsible for managing a set of specific meter instances. It provides a couple of convenience methods for interacting with meters, including a rich set of factory methods to easily construct new instruments. `Meter` is the abstract base class for all instruments that Micrometer offers.

## A `Counter` is a simple incrementing metric.

- Measure number of
  - requests
  - errors
  - completed tasks
  - ...

```
MeterRegistry registry = new SimpleMeterRegistry();

Counter counter = Counter
  .builder("request.incoming.total-amount")
  .description("Total # of incoming requests")
  .register(registry);

counter.increment();
counter.increment(2.0f);

assertThat(counter.count()).isEqualTo(3.0f);
```



## Any instrument supports multiple dimensions through tagging.

- Trait of the `Meter` base class
- A tag is a key-value pair
- Keys are re-used between instruments
- Values must be different

```
MeterRegistry registry = new SimpleMeterRegistry();

Counter getCounter =  registry.counter(
  "http.requests.total",     // ID
  "method",                  // key of the tag
  "GET");                    // value of the tag

Counter postCounter = registry.counter(
  "http.requests.total",     // ID
  "method",                  // key of the tag
  "POST");                   // value of the tag
```

## A Gauge represents a single numerical value that can increase or decrease over time.

- Measure
  - number of active threads
  - available memory
  - items in queue

```
MeterRegistry registry = new SimpleMeterRegistry();

// create a state object to be observed by the gauge
AtomicInteger queueSize = new AtomicInteger();

Gauge gauge = Gauge
  .builder("queue.size", queueSize, AtomicInteger::get)
  .description("Size of the work queue")
  .register(registry);

queueSize.set(5);

assertThat(gauge.value()).isEqualTo(5.0f);

queueSize.incrementAndGet();

assertThat(gauge.value()).isEqualTo(6.0f);
```



## A `Timer` measures the duration of operations and their frequency.

- Measure
  - duration of operation execution
  - response times
  - service times

```
MeterRegistry registry = new SimpleMeterRegistry();

Timer timer = Timer
  .builder("http.request.duration")
  .register(registry);

timer.record(() -> {
  try {
    TimeUnit.MILLISECONDS.sleep(500);
  } catch (InterruptedException e) {
    // ignore
  }
});

timer.record(300, TimeUnit.MILLISECONDS);

assertThat(timer.totalTime(TimeUnit.MILLISECONDS))
  .isGreaterThanOrEqualTo(800);
assertThat(timer.count()).isEqualTo(2);
```

## A `DistributionSummary` measures the distribution of a stream of values.

- Provide statistical information
  - sum
  - count
  - average
  - percentile values

```
MeterRegistry registry = new SimpleMeterRegistry();

DistributionSummary summary = DistributionSummary
  .builder("request.payload.size")
  .description("Payload size of HTTP requests")
  .tags("endpoint", "/api/data")
  .register(registry);

summary.record(500);
summary.record(1500);
summary.record(1000);
summary.record(2000);
summary.record(3000);

assertThat(summary.totalAmount()).isEqualTo(8000);
assertThat(summary.count()).isEqualTo(5);
```



## Expose

via Spring Boot Actuator

## Spring Boot auto-configures a composite `MeterRegistry` that is exposed via Actuator.

```
{
  "names": [
    "application.ready.time",
    "application.started.time",
    "disk.free",
    "disk.total",
    "executor.active",
    "executor.completed",
    ...
  ]
}
```

- Available at `/actuator/metrics`
- Inspect individual metric at `/actuator/metrics/{name}`



## Inject the `MeterRegistry` into a bean and register or fetch instruments by their name.

```
@Service
public class MyObservableService {

  private final Counter counter;

  public MyObservableService(MeterRegistry meterRegistry) {
    this.counter = meterRegistry.counter("my-operation-counter");
  }

  public void myObservableOperation() {
    /* some business logic */
    counter.increment();
  }
}
```

# Advanced Spring – Business-Related Metrics

## Provide custom Actuator endpoints for derived business metrics.

```
public class BusinessMetrics {

  @JsonProperty("revenue")
  private final double revenue;

  @JsonProperty("conversionRate")
  private final double conversionRate;

  public BusinessMetrics(double revenue,
                         double conversionRate) {
    this.revenue = revenue;
    this.conversionRate = conversionRate;
  }
}
```

```
@Component
@Endpoint(id = "businessMetrics")
public class BusinessMetricsEndpoint {

  /* declare instruments as members of this class */

  public BusinessMetricsEndpoint(MeterRegistry registry) {
    /* bind instruments of MeterRegistry to members */
  }

  @ReadOperation
  public BusinessMetrics snapshot() {
    return new BusinessMetrics(
      computeRevenue(),
      computeConversionRate());
  }
}
```

## Be aware that integrating this with a monitoring solution might require a dedicated format!

### Integration with Prometheus

- Prometheus scrapes service instances for metrics periodically
- Spring Boot exposes a dedicated Actuator endpoint
  - `/actuator/prometheus`
  - Exports metrics in Prometheus scrape format
- Module: `io.micrometer:micrometer-registry-prometheus`

# Store, query, and visualize

**A brief introduction to Prometheus and Grafana**

**So, we've exposed metrics per service instance. What about the big picture?**

- Not manageable to fetch metrics from service instances
  - Service instances come and go
  - How to interpret derived data?

**Requires central authority**

- ... for storing data points
- ... for querying and manipulating these data points

## Prometheus is a time-series database that offers a powerful query language.

1. Stores sequence of data points across time
2. Used to store metrics and performance data
3. Uses a pull-based approach to fetch metrics
4. Besides query operators, PromQL offers aggregations as well
5. Instruments are compatible with Micrometer

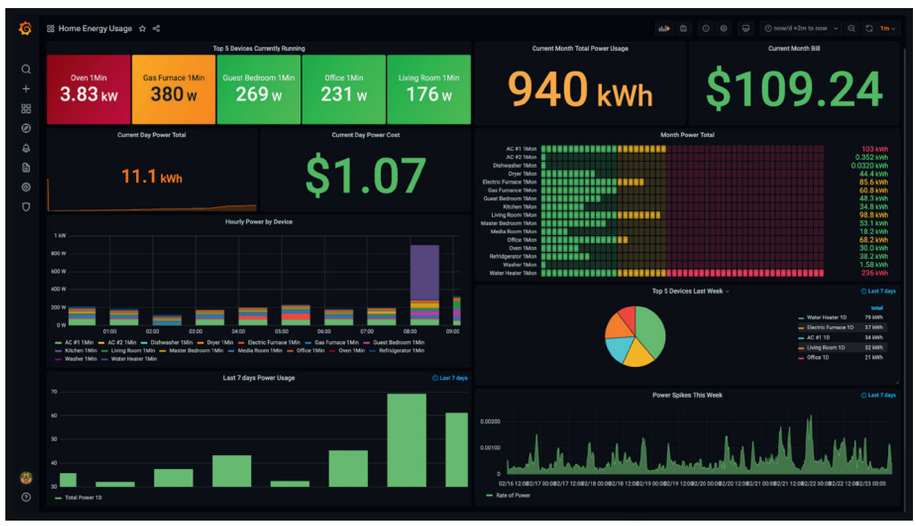


## PromQL is a nested functional language.

```
# Root of the query, final result, approximates a quantile.
histogram_quantile(
  # 1st argument to histogram_quantile(), the target quantile.
  0.9,
  # 2nd argument to histogram_quantile(), an aggregated histogram.
  sum by(le, method, path) (
    # Argument to sum(), the per-second increase of a histogram over 5m.
    rate(
      # Argument to rate(), the raw histogram series over the last 5m.
      demo_api_request_duration_seconds_bucket{job="demo"}[5m]
    )
  )
)
```

example from article *The Anatomy of a PromQL query* @ promlabs.com

**Grafana is a visualization tool that integrates with a variety of data sources.**



# Advanced Spring
## Workshop

**Boris Fresow** is a freelance IT consultant and trainer. He mainly deals with distributed systems, messaging solutions, and training on these topics.

**Markus Günther** is a freelance software developer, architect, and trainer. He supports his clients in implementing robust, scalable systems and also offers seminars & workshops on messaging solutions and modern software methodology.