Advanced Spring

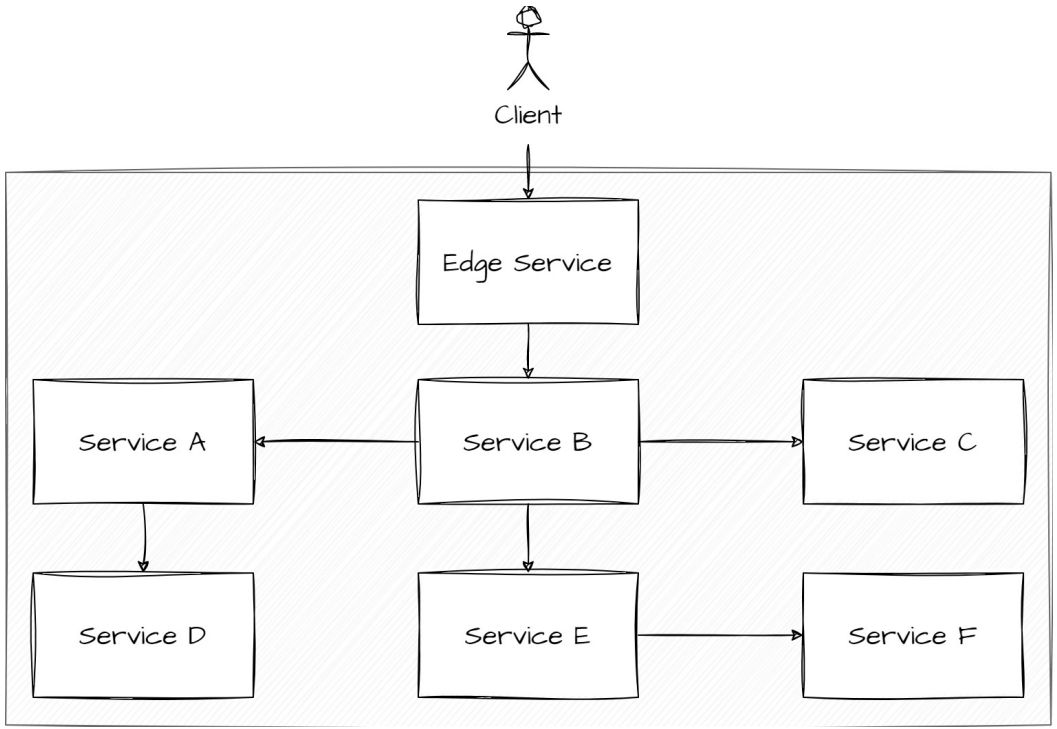
Edge Service with Spring Cloud Gateway

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.

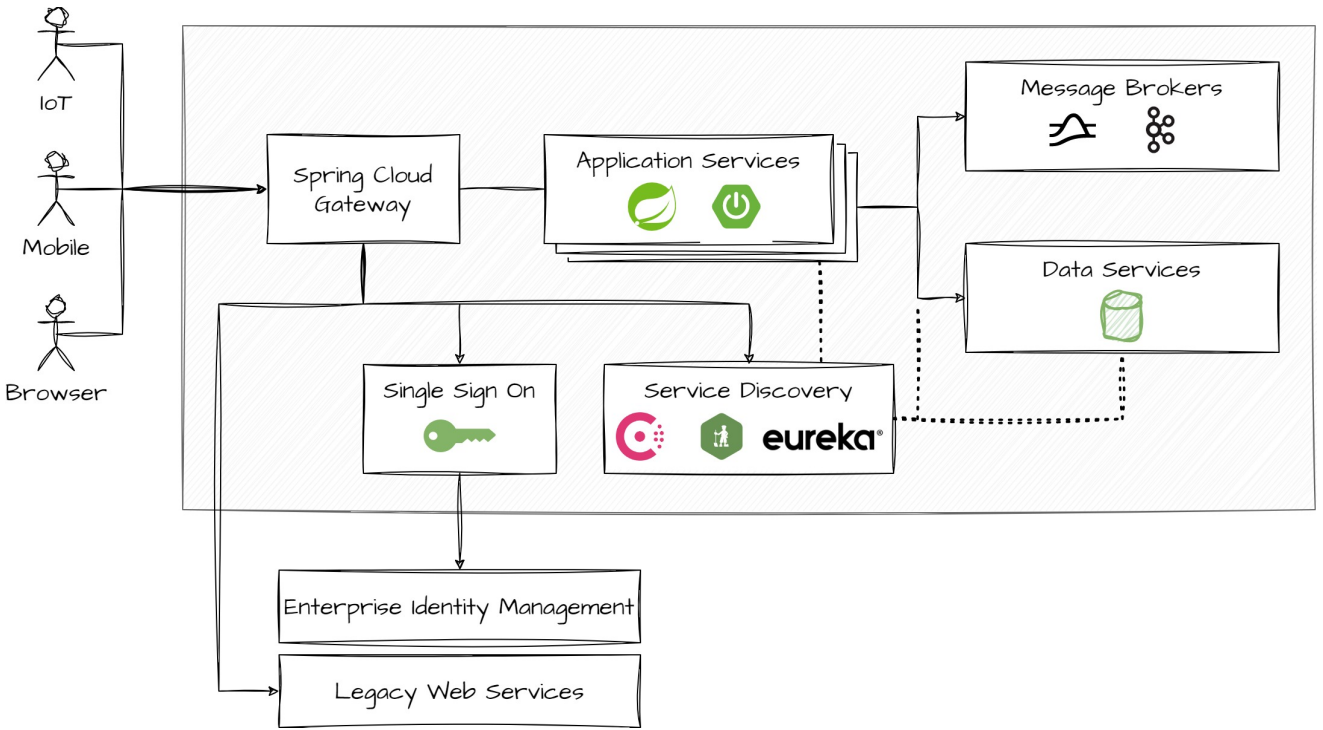
What are typical use cases for edge services?

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

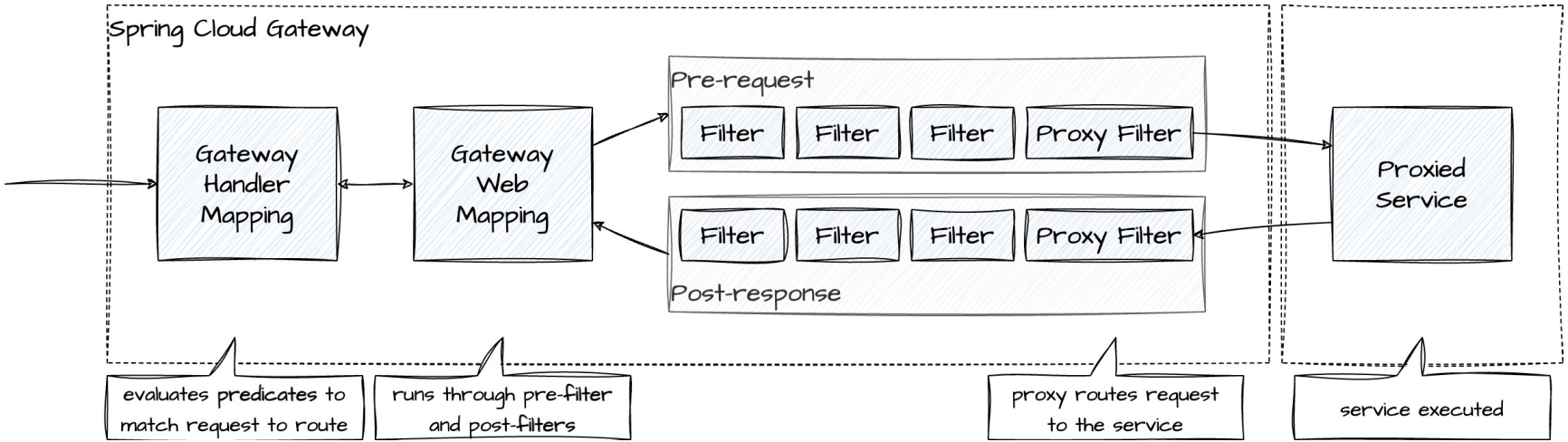
Use standard protocols and best practices such as OAuth, OIDC, JWT and API keys to ensure that clients are trustworthy.

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.

```
@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.

```yaml
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.

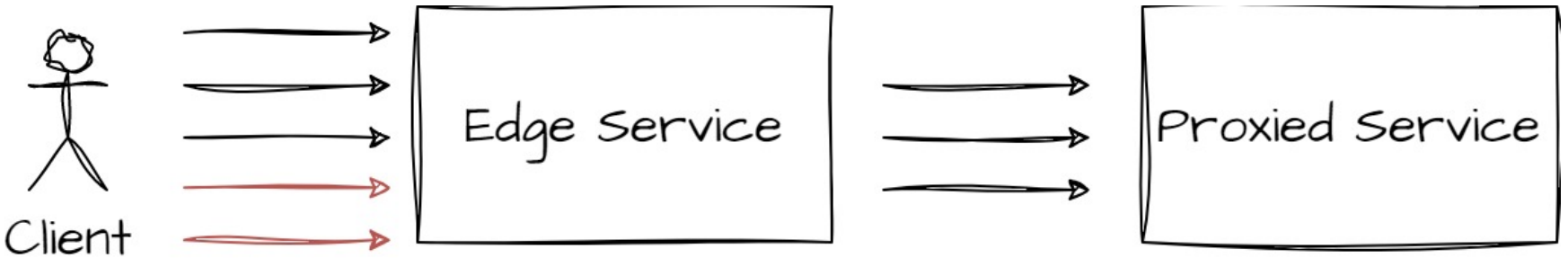
Rolling your own filter is a matter of implementing two interfaces.

- 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`.

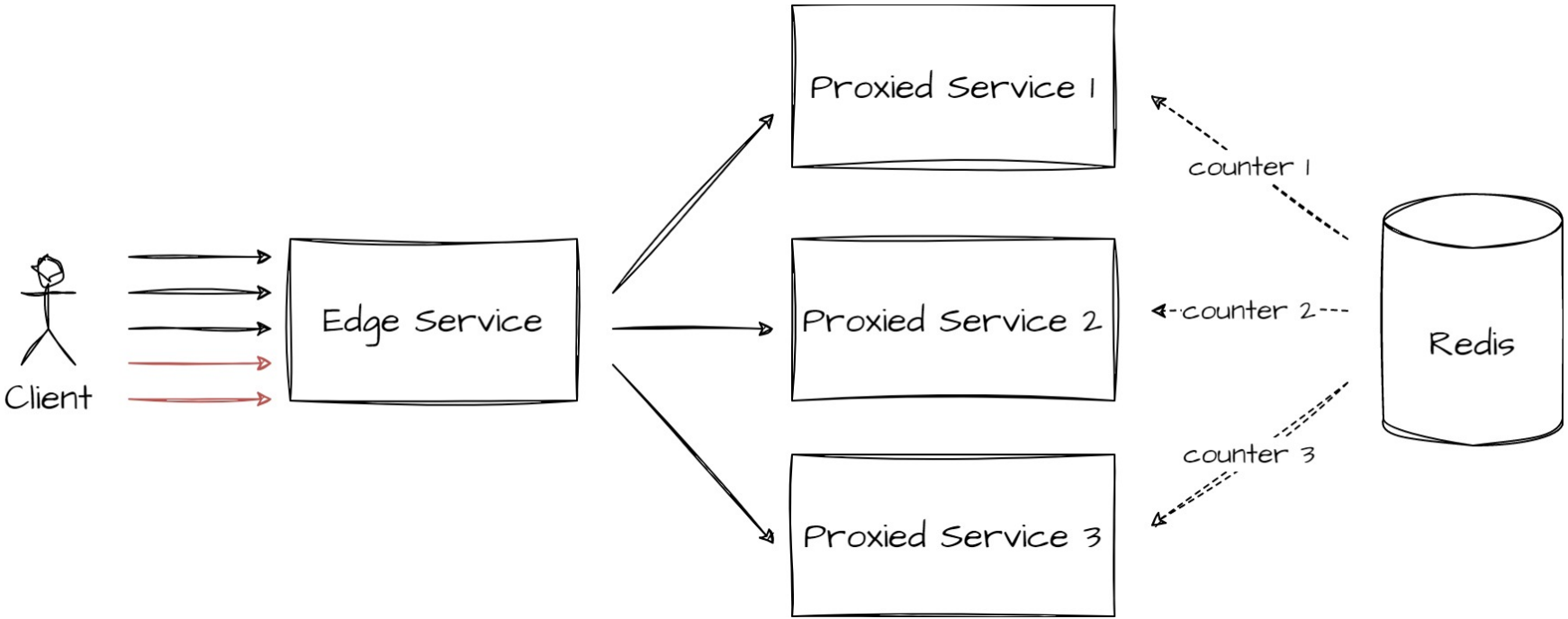
```
@Configuration
public class GatewayConfiguration {
  @Bean(name = "fixedIntervalRateLimiter")
  public FixedIntervalRateLimiter fixedIntervalRateLimiter() {
     return new FixedIntervalRateLimiter(Duration.ofSeconds(5), 1);
  }

  @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}"
```

Reference beans as SPEL expression.

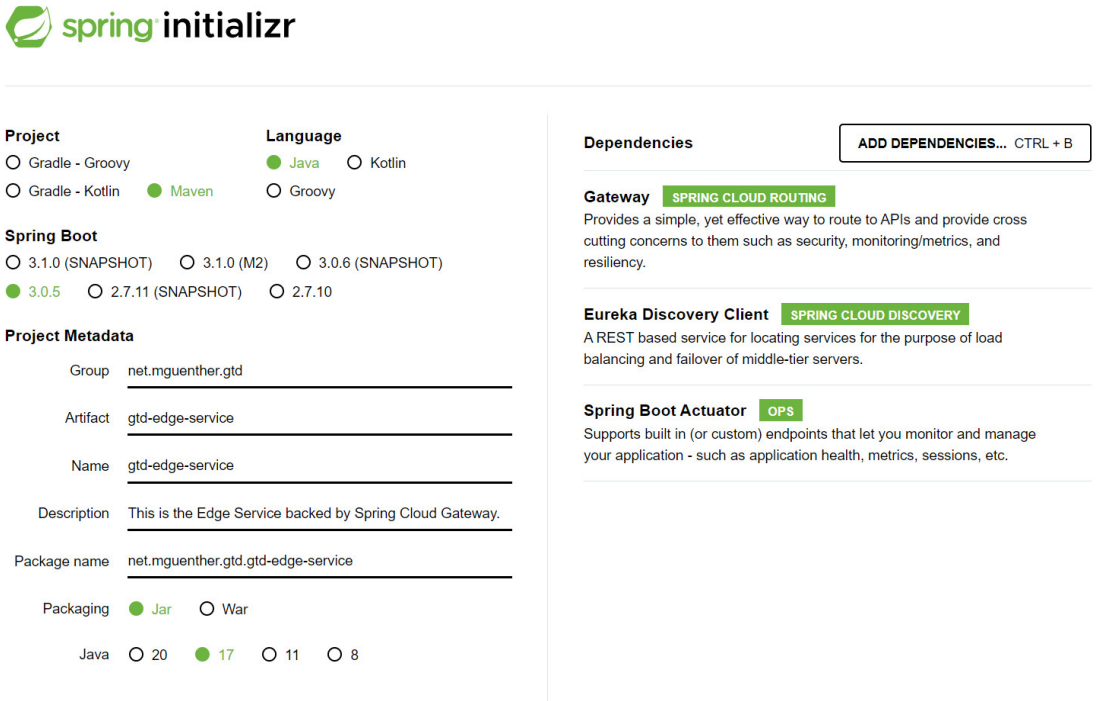
A production-grade rate limiting strategy requires distributed counters.



Spring Cloud Gateway integrates well with Redis. Redis supports distributed counters. This allows you to implement a token bucket strategy.

Setting up an Edge Service with Spring Cloud Gateway

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



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

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.

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

Especially `management.endpoint.gateway.enabled` is important, as this lets you live-debug all routes that are currently configured at the edge service.

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

```java
@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"
        }
    }
}
```

Questions?