Architecture vision and design

KRIS5

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

Change history

Date	Ver	Description	Author
06.04.18	0.1	Template and first contents	Andres Aavik
12.04.18	0.2	Design principles added	Andres Aavik
16.04.18	0.3	Microservice principles	Andres Aavik
17.04.18	0.4	API Gateway, Service Discovery	Andres Aavik
18.04.18	0.5	User interface components	Andres Aavik
19.04.18	0.6	Added graphics, logs	Andres Aavik
24.04.18	0.7	Authentication, data integrity, message broker, non-functional requirements	Andres Aavik
02.05.18	0.8	Added 5 main description; added 3.4 “Synchronous or asynchronous interaction”; Review of whole document	Andres Aavik
22.05.18	0.9	EU logo	Andres Aavik
25.05.18	1.0	Added missing English translations	Andres Aavik

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

Table of contents

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

1 Introduction

This document consists:

- Vision (E1)
- Architecture design principles (E4)
- User interface components (E2, E5)
- Back-end (E2, E5, E8, E10)
- Scaling and backup (E3, E9)
- Non-functional requirements (E6)

This document does not contain:

- Economical aspects (E7)
- Mock-up design and demo (E11)
- Visualization of the demo (E12)

2 Vision

The proposed technical solution for the Systems Architecture of KRIS5 is a distributed system that is easily expandable as well as scalable. The solution is put together from open source components and frameworks avoiding any added cost for licencing.

The system will mainly be created using Java (back-end) and Javascript / Typescript (user interface components) programming languages. The solution would consist of a total of four component groups: user interface components (front-end), back-end microservices, databases and monitoring/control/support systems. The component groups are loosely interconnected and thus capable of functioning on completely different servers and/or containers, can be editable or interchangeable independently of one another.

The core technical system for the back-end would be the Spring Cloud, Spring Framework components-based framework. One of the main objectives of this framework is to offer the widest possible separation between the various components using the microservices and the REST APIs, making it a good tool for creating state-of-the-art applications. The back-end system is scalable with the Ribbon component. The Spring Cloud architecture is depicted in Figure 1.

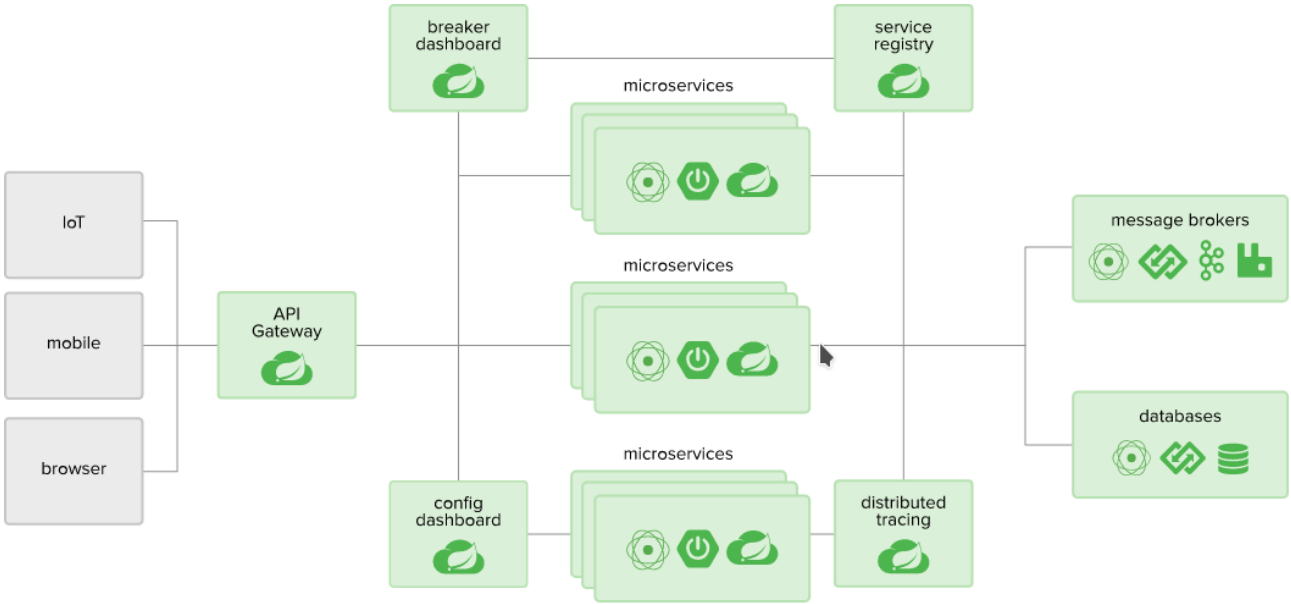


Figure 1. Spring Cloud general architecture concept

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

As a database, PostgreSQL is offered as a powerful alternative to paid database solutions. For PostgreSQL, databases can be replicated using Streaming Replication technology to ensure that data is retained in the event of a hardware or software failure. If necessary, it is possible to use different databases (Oracle, MSSQL, noSQL) and various databases as a collection.

The front-end (user interface) would be created using the Bootstrap framework. Either Angular or a suitable JavaScript or Typescript based front-end framework (React, Vue.JS) should be chosen based on available competences and RIK general front-end architecture strategy. The goal is to create an interface using responsive web design and Progressive Web Apps. The connection between the front-end and the back-end would work over the REST API.

No connection to any database or service was ever performed directly from the front-end system, and the only component to communicate with is a back-end system which in turn provides all the necessary services. Also, all operations with data and business logic decisions (recording, validation, etc.) would always be done in the back-end system. The only difference would be very low level controls in the user interface components (for example: the required fields must be filled in to fill out the form; data format – date, number etc.).

3 Architecture design principles

The proposed architecture vision with all the components is depicted on Figure 2.

3.1 MICROSERVICES DESIGN

Microservices should have the following main characteristics:

- Single responsibility;
- Deployable independently;
- Scalable independently;
- Small in size;
- Easy to replace.

These characteristics are needed to keep in mind in designing microservices.

Microservices should be designed and implemented based on the following criteria:

1. Business services - part of business process or business domain, i.e. application document registration, application document processing.
2. Integration services – services to get data or send data to various third party systems, i.e. X-road services.
3. Support services – processes that can be used by business processes i.e. PDF generation, message service (i.e. email, SMS) or used by all the services (i.e. authentication and authorization).
4. User Interface Components –user interface applications for various devices (i.e. web, mobile, tablet) and users (i.e. citizens, enterprises, public sector employees).

Questions to ask in design process. Depending on the answers design principles should be considered.

- What roles and users can access different services?
    - Microservices do not have its own security layer as the usage of central authentication and authorization is used.
- Does the service have some peaks in usage?
    - If the service has peaks in usage, then the microservice should have its own database server.
- Does the service have high read or write actions in data persistence layer?
    - If the service has high write or read loads to data layer, then for scalability, it should have its' own database server.
    - The database could be different type, i.e. relational, nosql. If you have to replicate the database to different location, then use one type of database, i.e. relational PostgreSQL.
- Does the service have to save data in one transaction or multiple?
    - If the service is synchronous then the microservice should save persistent data in one transaction.
    - Consider to use one transaction per microservice and not to deal with multiple microservice transaction. If not possible, use like asynchronous service design.

- Does the service give the result synchronously or asynchronously?
    - Choosing between synchronous or asynchronous interaction depends on, if the user interface component needs the result in short time span or, where the user can wait for the result. [See **Tõrge! Ei leia viiteallikat.**]
    - If the service uses asynchronous service, then the data should be persisted, used data status (pending, active, deleted). The data rows should have status fields do distinguish from pending, active, deleted data. If data rollback occurs, you do not delete the data but mark it deleted. The microservice must have rollback services to mark data active or deleted.
    - Asynchronous services should have a message bus service. [See 5.5]
    - Asynchronous service should push data to user interface components through WebSockets technology.
- Does the service require different programming language to fit the solution?
    - Microservices could be designed to use different programming languages, as the fit the best performance for its purpose and tasks. The decision could be made in following:
        - There is an usable service solution in public sector
        - There is an usable software library in other language
        - Performance, some frameworks or programming languages might give better results in specific tasks, i.e. using R or Python for statistics or machine learning
    - The key is that the microservice can be used as REST API and all the proposed architecture can be used, i.e. security, load balancing.
- Does the business need reports or statistics or raw data from different microservices?
    - API Composition - the application performs the join rather than the database
    - Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each service publishes when it updates its data.[1]

[1] http://microservices.io/patterns/data/database-per-service.html

3.2 DATA PERSISTENCE

Database layer design key principle is to hold persistent data private for every microservice and let other microservices access the data through its API. There are exceptions for aggregating data, i.e. in reporting or opendata.

Choose the data persistence depending on the service design, security requirements and estimated need for scalability:

- Private tables – service has one or more private tables that are private (other services are unable to write) to that service.
- Schema-per-service – service has a database schema that's private to that service.
- Database-server-per-service – service has its own database server.

3.3 KRIS5 MICROSERVICES

Functional requirements should be analysed as business processes to see how microservices should be designed.

Functional requirements' based business proposed microservices:

- Application registration
- Proceeding details – appointment of a registrar, merging the proceedings, proceeding term, proceeding
- Operations – should be analysed in depth: which of all the processes can be split into multiple services and grouped by the nature and type of business process
- Rulings – should be analysed in depth: which of all the processes can be split into multiple services and grouped by the nature and type of business process
- Ex officio proceedings
- Certificates
- Inquiries and printouts
- Statistics and reports

Support services:

- API gateway
- PDF generation
- Scheduler
- Messaging – email, by post
- Qualified Digital Signature Service
- Autonomous ADS
- Authentication and authorization
- File management
- User management
- Classificator management
- Log management
- Data integrity and audit log
- Localization (i10n) and internalization (i18n) services
- Template and text management
- XML generation and management

- BPMN and RPA based process automation

Integration services (brackets include Estonian translation):

- Official publication Ametlikud Teadaanded (AT teated)
- Opendata (Avaandmed) - based on EU policy[2] and Estonian Green Book on opendata[3].
- X-Road query (X-tee päringud)
- X-Road provider (X-tee pakkuja)
- Land registry (Kinnistusraamat) old system KAEP – will be integrated to new system
- LRI, the EU Land Registers Interconnection service (üleeuroopaline kinnistusraamatute päringusüsteem)
- E-Notary system (E-notar)
- Third party authentication and authorization services (*kolmandate osapoolte autentimise ja autoriseerimise teenused*)

User Interface Components (could be also combined and depends on provided services):

- Public e-land register
- Citizen e-land register
- Enterprise e-land register
- Public servant e-land register
- Reporting

3.4 SYNCHRONOUS OR ASYNCHRONOUS INTERACTION

The interaction facet is about whether two services need to be available at the same time in order to successfully interact. Tight coupling is implied by synchronous interactions, which require both parties to be available at the same time in order to communicate. Loose coupling is typically associated with asynchronous interactions, where a successful interaction can happen even if one of the involved parties is not available at the same time.

In service-oriented systems, the interaction facet of loose coupling plays a major role in order to enable the communications of subsystems that are provided as a service by a different organization than the one consuming them. Thanks to the properties of asynchronous message-based communications, it becomes possible to remove the time dependencies between both ends of the communication.

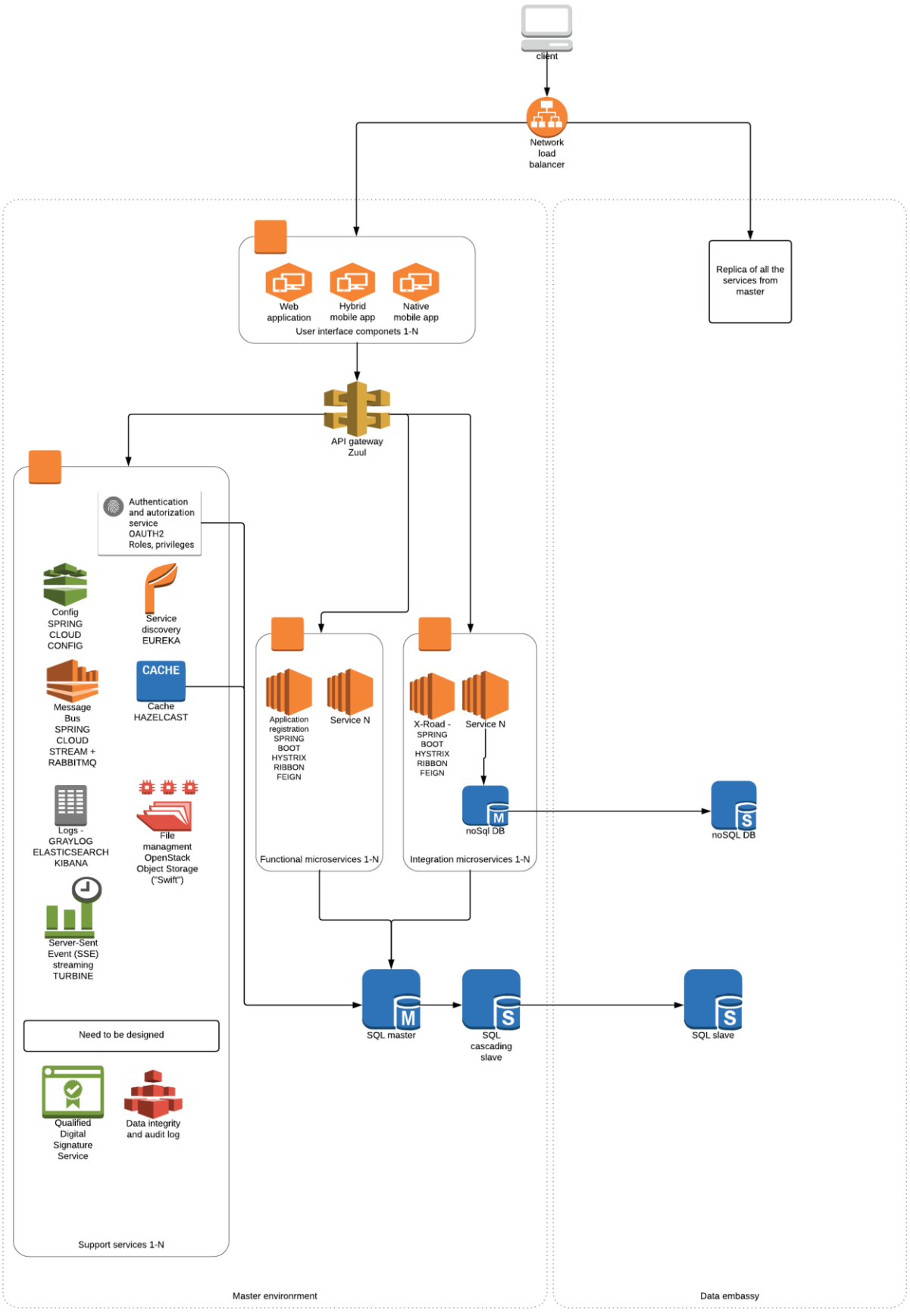
For example, when it comes to performing maintenance tasks on the services, as these do not always need to be available to immediately handle client requests, the service providers do not need to schedule outages taking into account the needs of their clients. On the other hand, a service that is published using a synchronous communications protocol requires a bigger investment to avoid outages in the

[2] Digital Single Market, POLICY, Open data portals https://ec.europa.eu/digital-single-market/open-data-portals
[3] https://opendata.riik.ee/et/roheline-raamat

provider infrastructure, as client request messages will be lost if the service supposed to process them becomes (even temporarily) unavailable.[4]

[4] ”Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design” C. Pautasso, E. Wilde http://www2009.eprints.org/92/1/p911.pdf


European Union
European Regional
Development Fund
Investing
in your future


client
Network
load
balancer
Web
application
Hybrid
mobile app
Native
mobile app
User interface componets 1-N
Replica of all the
services from
master
API gateway
Zuul
Authentication
and autorization
service
OAUTH2
Roles, privileges
Config
SPRING
CLOUD
CONFIG
Service
discovery
EUREKA
Message
Bus
SPRING
CLOUD
STREAM +
RABBITMQ
CACHE
Cache
HAZELCAST
Logs -
GRAYLOG
ELASTICSEARCH
KIBANA
File
managment
OpenStack
Object Storage
("Swift")
Server-Sent
Event (SSE)
streaming
TURBINE
Need to be designed
Qualified
Digital
Signature
Service
Data integrity
and audit log
Support services 1-N
Application
registration
SPRING
BOOT
HYSTRIX
RIBBON
FEIGN
Service N
Functional microservices 1-N
X-Road -
SPRING
BOOT
HYSTRIX
RIBBON
FEIGN
Service N
noSql DB
Integration microservices 1-N
noSQL DB
SQL master
SQL
cascading
slave
SQL slave
Master environrment
Data embassy

4 Technical solution for user interface components

The technical solutions, standards, frameworks and libraries for user interface components are described in this section. The list is not comprehensive and every user interface component must be analysed and designed for specific user need and also devices used by the user. User interface components are packaged as separate applications from the back-end and use APIs to communicate with the back-end. As they are designed as microservices, then these applications can be scaled as other microservices used in back-end.

The web application design should be towards Progressive Web Apps (PWAs)[5]. PWAs are defined by a set of concepts and keywords including progressive, responsive, connectivity independent, app-like, fresh, safe, discoverable, reengageable, installable, and linkable. The current state of progressive web apps involves a lack of certain hardware and platform APIs and features that only (certain) cross-platform and native apps can access.[6]

Hybrid apps should be considered if there is a need for using standard mobile properties, like contacts or camera. A hybrid app is essentially a small website written with HTML, CSS, and JavaScript. It is different from normal websites in that it runs only in a browser shell and has access to the native platform layer. To run like a native app, it relies on a native wrapper like Cordova.[7]

Native mobile and tablet solutions should be considered, as there is a need for specific device properties that hybrid frameworks do not include, i.e. Cordova platform support.[8]

4.1 STANDARDS AND BEST PRACTICES

These standards or best practices should be used as the basis of web based components:

- TypeScript[9] version 2 or later
- Javascript[10] based on ECMAScript 2015 (ES6) or later
- HTML 5 – designed for grid layout and responsive web;

[5] https://developers.google.com/web/progressive-web-apps/?hl=en
[6] ”Progressive Web Apps: The Possible Web-native Unifier for Mobile Development” Andreas Biørn-Hansen, Tim A. Majchrzak and Tor-Morten Grønli http://www.scitepress.org/Papers/2017/63537/63537.pdf
[7] ”Hybrid app approach: could it mark the end of native app domination?” M Huynh, P Ghimire, D Truong http://iisit.org/Vol14/IISITv14p049-065Huynh3472.pdf
[8] https://cordova.apache.org/docs/en/latest/guide/support/
[9] https://www.typescriptlang.org/
[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

- CSS 3 – designed for multimedia, grid layout and responsive web.

4.2 FRAMEWORKS

Web based application frameworks:

- Angular https://angular.io/ – TypeScript
- Vue.js https://vuejs.org/
- React https://reactjs.org/

Hybrid mobile and web applications:

- Ionic https://ionicframework.com/
- Onsen UI https://onsen.io/

4.3 HTML, CSS AND JS LIBRARIES

- Bootstrap version 4+[11] – Bootstrap is an open source toolkit for developing with HTML, CSS, and JS.
- HTML5 Boilerplate[12] - HTML5 Boilerplate helps you build fast, robust, and adaptable web apps or sites.
- Sass[13] - Sass is the most mature, stable, and powerful professional grade CSS extension language in the world.
- Font awesome[14] – Font Awesome gives you scalable vector icons that can instantly be customized — size, color, drop shadow, and anything that can be done with the power of CSS.

Package managers:

- npm[15] - npm is the package manager for JavaScript and the world's largest software registry.
- Yarn[16] - Yarn is a package manager for your code.
- Webpack[17] - At its core, webpack is a static module bundler for modern JavaScript applications.

Quality assurance:

- Karma[18] - Karma is essentially a tool which spawns a web server that executes source code against test code for each of the browsers connected.
- Protractor[19] - Protractor is an end-to-end test framework for Angular and AngularJS applications.

---

[11] https://getbootstrap.com/
[12] https://html5boilerplate.com/
[13] https://sass-lang.com/
[14] http://fontawesome.io/
[15] https://www.npmjs.com/
[16] https://yarnpkg.com
[17] https://webpack.js.org/
[18] https://karma-runner.github.io/2.0/index.html
[19] http://www.protractortest.org

- Jasmine[20] - Jasmine is a behaviour-driven development framework for testing JavaScript code.

[20] https://jasmine.github.io/

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

5 Technical solution for back-end

Back-end architecture is based on Spring Cloud, Spring Cloud Netflix (Netflix Open Source Common Runtime Services & Libraries) and other compliant solutions. The solutions chosen are open-source, well-documented and the activity of development is still ongoing.

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.[21]

Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).[22]

The solution uses these software libraries:

- Zuul - API gateway
- Eureka – service discovery
- Spring Cloud Config – configuration management
- Spring Cloud Security – authentication and authorization
- Spring Cloud Stream and RabbitMQ – message broker
- Graylog – log managent
- Guardtime KSI Blockchain – date integrity
- Hazelcast - cache
- Hystrix – fault tolerance
- Ribbon – client side load balancing
- Feign – Java to HTTP client binder
- OpenStack Object Store Swift – data storage
- PostgreSQL – database

[21] http://projects.spring.io/spring-cloud/
[22] https://cloud.spring.io/spring-cloud-netflix/

5.1 API GATEWAY (ZUUL)

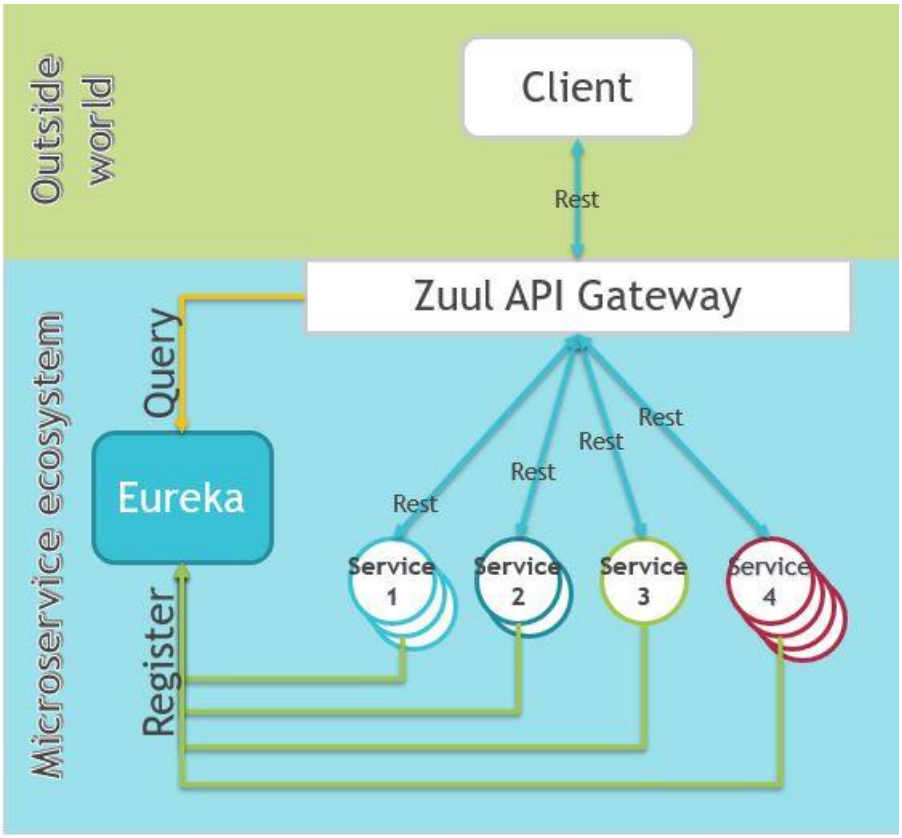


Figure 3. Zuul main architecture

Zuul is an API gateway, also known as an Edge service. It receives all the requests from different clients (either UI, mobile application, external service or similar) and redirects them to internal microservices. Being a microservice itself, it can be scaled and deployed independently and thus any services requiring centralization can be implemented and managed at one single point and do not need to be included into every single microservice. At Netflix, where Zuul was created to handle the volume and diversity of their API traffic, the following functions are used:

1. **Authentication and Security - identifying authentication requirements for each resource and rejecting requests that do not satisfy them.**
2. **Insights and Monitoring - tracking meaningful data and statistics at the edge in order to give us an accurate view of production.**
3. **Dynamic Routing - dynamically routing requests to different back-end clusters as needed.**
4. Stress Testing - gradually increasing the traffic to a cluster in order to gauge performance.

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

5. Load Shedding - allocating capacity for each type of request and dropping requests that go over the limit.
6. Static Response handling - building some responses directly at the edge instead of forwarding them to an internal cluster
7. Multiregion Resiliency - routing requests across AWS regions in order to diversify our ELB usage and move our edge closer to our members

Most of the functionality would immediately benefit the KRIS5 functionality but later once loads and requirements have changed, it could prove very difficult to implement Zuul or a similar service as all the microservices would have to be modified. However even functions 1-3 in the list above justify including Zuul in the microservice architecture design of KRIS5.

Link: https://github.com/Netflix/zuul/wiki

5.2 SERVICE DISCOVERY

Eureka Server is a REST based service for locating microservices for the purpose of load balancing and failover of middle-tier servers. The Eureka Client is a Java based client component which simplifies communication with the Eureka Server. The need for such a mid-tier service registry comes from the fact that in a dynamic environment, the services change states and form based on situational requirements. On a side note, when using Eureka, it is suggested to keep all services stateless for better scalability and compatibility. The high level overview of multi-zone Eureka deployment at Netflix depicted on Figure 4.

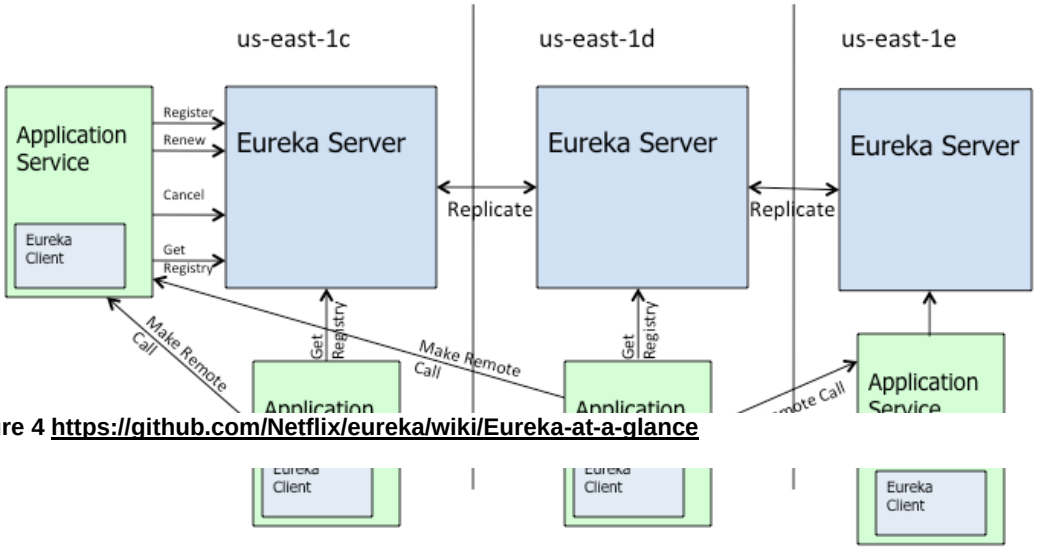


Figure 4 https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Registering

When a client registers with Eureka, it provides meta-data about itself, such as host and port, health indicator URL, home page, etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

Status Page and Health Indicator

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

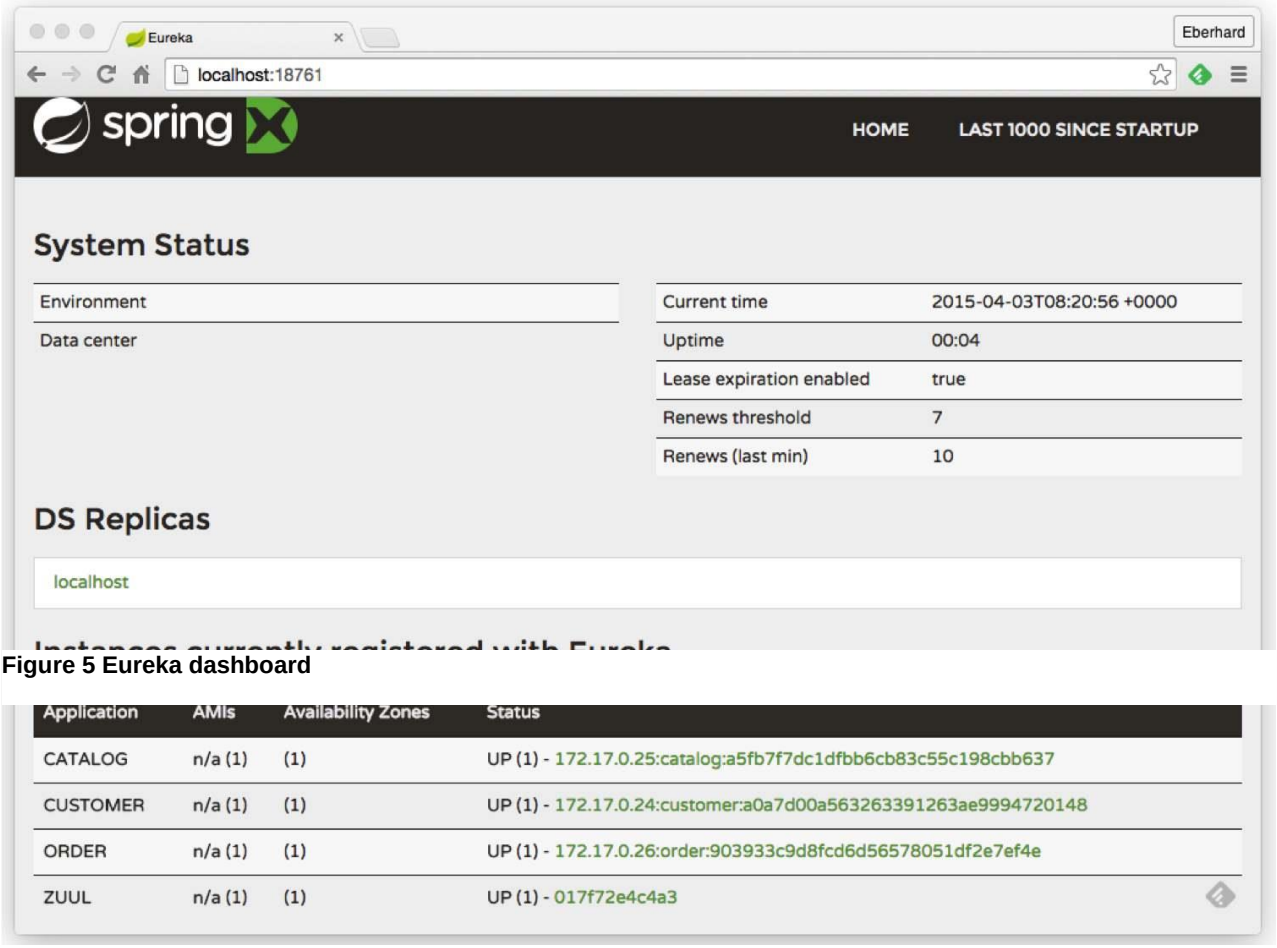


Figure 5 Eureka dashboard

Eureka's Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise the Discovery Client will not propagate the current health check status of the application per the Spring Boot Actuator. Which means that after successful registration Eureka will always announce that the application is in 'UP' state. Enabling Eureka health checks can alter this behaviour, which results in propagating application status to Eureka. As a consequence, every other application won't be sending traffic to the application in state other than 'UP'.

Link: https://github.com/Netflix/eureka/wiki

5.3 CONFIGURATION (SPRING CONFIG)

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring *Environment* and *PropertySource* abstractions, so they fit very well with Spring applications, but can be used with any application running in any language. As an application moves through the deployment pipeline from development to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage back-end uses git so it easily supports labelled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

Link: http://cloud.spring.io/spring-cloud-static/spring-cloud-config/2.0.0.M9/single/spring-cloud-config.html

The Config servers:

- Can work in an active-active or active-passive cluster configuration if a simple load balancer is used or Config server is registered in the Eureka Service Discovery.
- Support multiple code repositories for storing configuration data.
- Support file store for storing configuration data.
- Password encryption and key management.

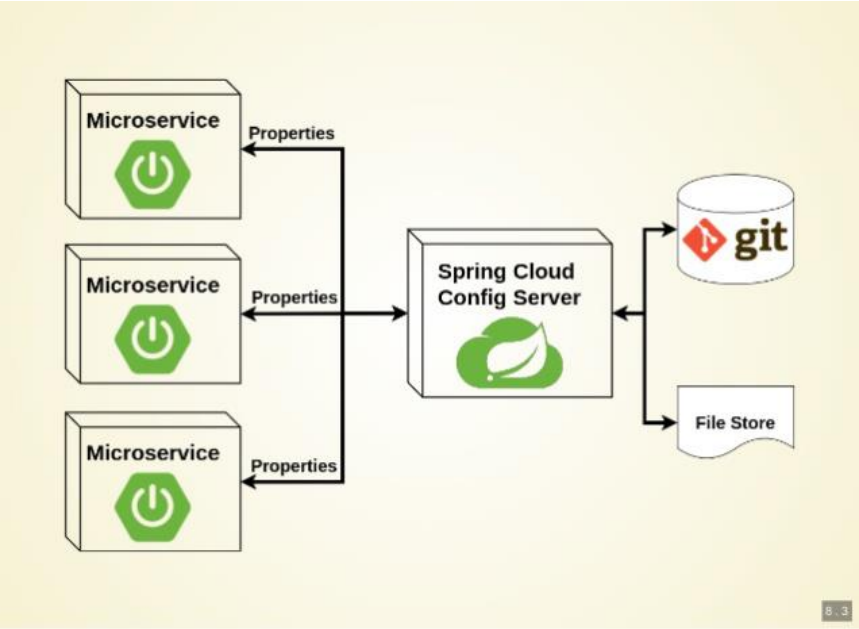


Figure 6. Spring Cloud Config Server architecture view [source: https://mromeh.com/2017/12/04/spring-boot-with-embedded-config-server-via-spring-cloud-config/]

5.4 AUTHENTICATION AND AUTHORIZATION

The authentication consists of two parts: public and microservice.

Public authentication is for using in user interface components. Modules: ID-card, Mobile-ID, Smart-ID or third party authentication systems through integration services like STORK, Active Directory and Bank links. It uses also OAuth2 but also OpenID can be used.

Microservice authentication authenticates services inside back-end. It is based on OAuth2.

Authorization is role based. Every role has its privileges. Microservice authorization can be also microservice-based. It means, only role is system and every API call is allowed to that microservice.

For security, Spring Cloud Security could be used. It makes SSO and OpenID authentication available. Spring Cloud Security features:

- Relay SSO tokens from a user interface component to a back-end service in a Zuul proxy.
- Relay tokens between resource servers.

- An interceptor to make a Feign client behave like OAuth2RestTemplate (fetching tokens etc.).
- Configure downstream authentication in a Zuul proxy.

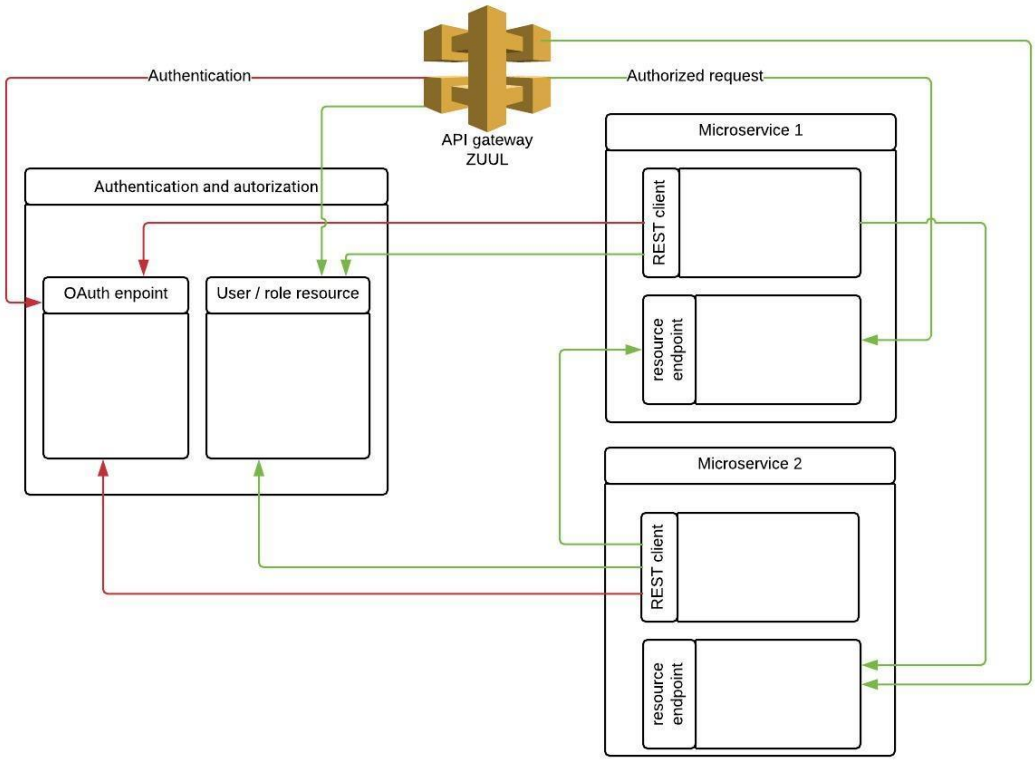


Figure 7. Authenication in API and microservices

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

5.5 MESSAGE BROKER

In computer programming, a message broker is an intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication or computer networks where software applications communicate by exchanging formally-defined messages. Message brokers are a building block of message-oriented middleware.[23] See Figure 8.

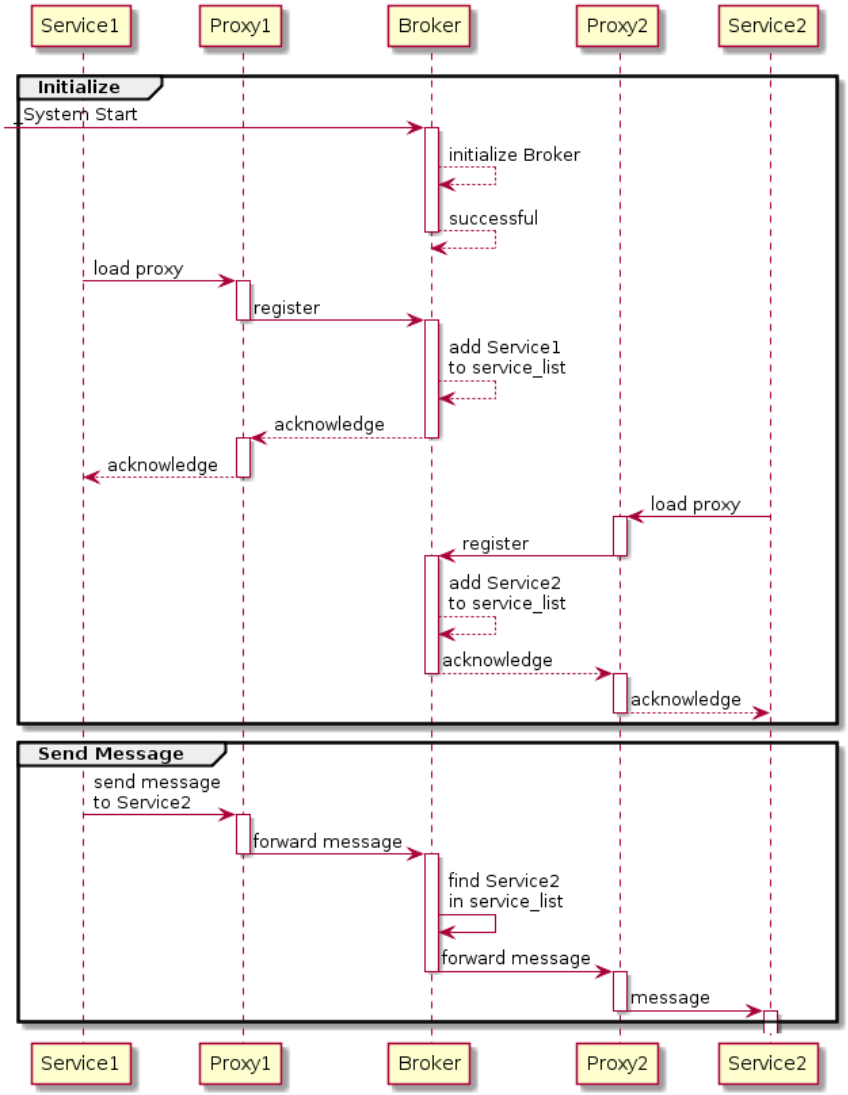


Figure 8. Sequence diagram for depicting the Message Broker pattern

[23] https://en.wikipedia.org/wiki/Message_broker

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications, and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.[24]

Spring Cloud Stream provides a number of abstractions and primitives that simplify the writing of message-driven microservice applications. Spring Cloud Stream provides Binder implementations for Kafka and Rabbit MQ. Spring Cloud Stream also includes a TestSupportBinder, which leaves a channel unmodified so that tests can interact with channels directly and reliably assert on what is received.[25]

For KRIS5 solution RabbitMQ is chosen.

RabbitMQ is an open source message broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQTT, and other protocols.[26]

A simplified diagram of how the RabbitMQ binder operates can be seen

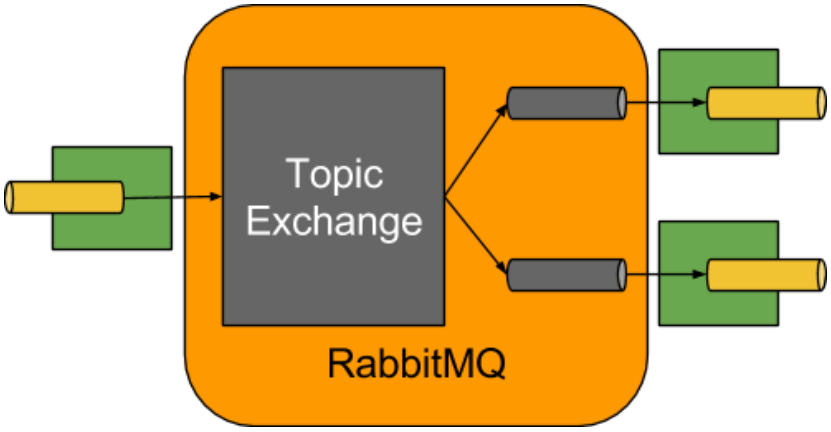


Figure 9. RabbitMQ Binder

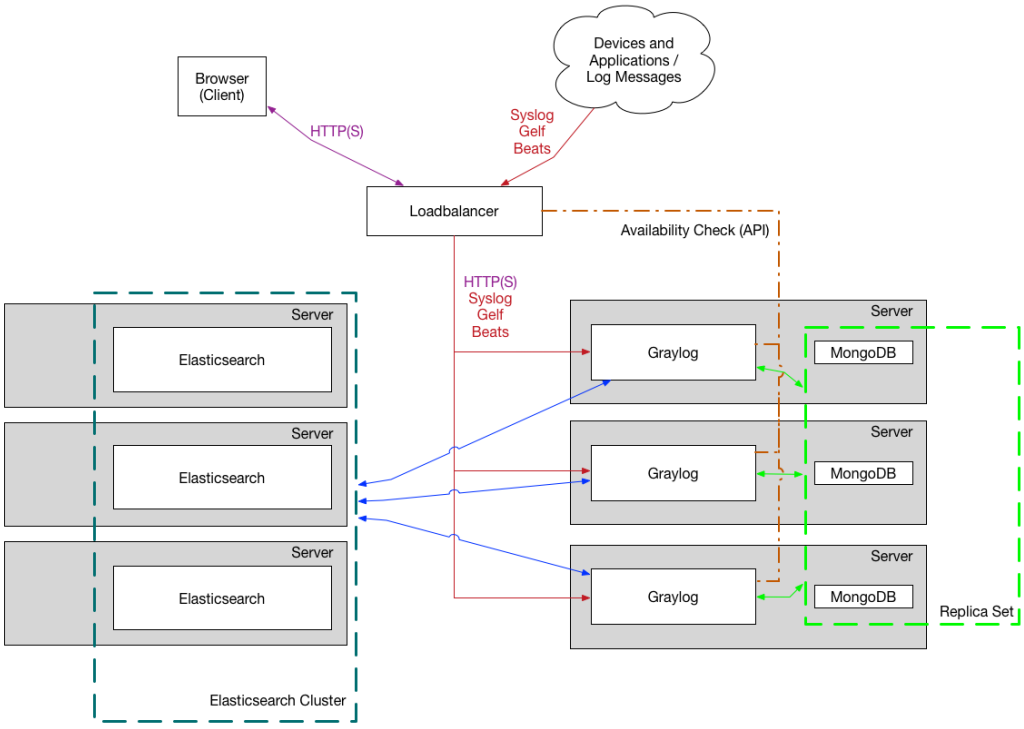
The RabbitMQ Binder implementation maps each destination to a TopicExchange. For each consumer group, a Queue will be bound to that TopicExchange. Each consumer instance have a corresponding RabbitMQ Consumer instance for its group's Queue. For partitioned producers/consumers the queues are suffixed with the partition index and use the partition index as routing key.[27]

[24] https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_introducing_spring_cloud_stream
[25] https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_main_concepts
[26] https://en.wikipedia.org/wiki/RabbitMQ
[27] https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_rabbitmq_binder_overview

5.6 LOGS

GRAYLOG

Graylog is an open source log management, aggregation and search tool. It has powerful customization options, can be integrated with various log providers and is based on scalable architecture. Please see Figure 10 for large scale production setup.



As for input Graylog accepts:

- syslog,
- Apache Kafka transport (GELF, syslog, raw/plaintext),
- RabbitMQ (AQMP) transport queue,
- HTTP API (JSON)

There is also a wide selection of plugins in a dedicated Graylog marketplace.

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

5.7 DATA INTEGRITY

Firstly, the data integrity must be designed and these solutions proposed in this section analysed.

Data layer should use event sourcing and Command Query Responsibility Segregation (CQRS) pattern. CQRS definition: “The separation of application or system responsibilities into Writing and Reading at overall architectural level rather than internal object level”[28]. Also Domain-driven design (DDD)[29] could be used for designing the data persistence layer. The data UPDATE and DELETE actions in database is prohibited in such models. Every data change should have its own row in database and event rows in common event table or if needed, for every table or tables used in one transaction.

One solution for using event sourcing and CQRS is Eventuate™ Local[30]. Downside is that this solution has only Apache Kafka support for message broker. Alternative solution is to design special solution with blockchain solution in mind.

For data integrity, timestamping and/or blockchain solution must be used. The ledger service could be built by using some kind of open source blockchain solutions or use enterprise grade solutions like Guardtime KSI Blockchain.

Every data entity should have columns: entity id, foreign key ids to entities in current microservice 1-N, data columns 1 – N, metadata columns 1-N.

Metadata consists of at least:

- timestamp row created;
- creator user id;
- row hash;
- previous row hash.

Event data entity columns must be designed to comply with solution chosen.

To accomplish data integrity is as follows:

1. Every row hash should be calculated before commit and use all data and metadata fields. The hash is saved with the row and committed.
2. Every event row should have also a hash calculated from connected data hash and event data row. The event data is committed and published to message broker.
3. Client side data aggregation and hash calculation should be used. One solution is to use the same mechanism as blockchain solution – aggregation tree. This is needed for performance and avoiding so called “bottle-necks”.

---

[28] ”Practitioners’ view on command query responsibility segregation” Nazife Korkmaz, Martin Nilsson
http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=4864802&fileOId=4864803

[29] https://en.wikipedia.org/wiki/Domain-driven_design

[30] https://github.com/eventuate-local/eventuate-local

4. Send aggregated hash to blockchain solution through message broker and persist the timestamped signature hash.

The key is to not send data to ledger but data hashes or data aggregated hashes to blockchain solution.

5.8 CACHE

A cache transparently stores data so that future requests for that data can be served faster. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location such as a database, mainframe or service API, which is slower.

Hazelcast in-memory data grid is chosen for the architecture of KRIS5. The proven solution will suit even the most complex cases and can reduce response times for read requests from 5-50 times dependent on solution and implementation. Hazelcast also comes with a management centre which enables to monitor and manage members that are running Hazelcast IMDG. In addition to monitoring the overall state of the clusters, one can also analyse and browse your data structures in detail, update map configurations, and take thread dump from members.

As with most complex solutions, the configuration and setup is the key. It is essential that the Hazelcast configuration is fine-tuned in pair with performance tests before every major project release to production. Some main points to note in addition:

- Use client-server architecture (See Figure 11) which is suitable for high-security environments and where a) different clients may have different security policies b) multiple applications will access the same cache pool (there may be an unknown number of UI blocks for the architecture) c) both front- and back-end systems will be clustered.
- The most common access patterns are read-through, write-through, and write-behind. Careful thought must be given to choose the most suitable solution from read-through and write-through access patterns. Write-behind is not recommended.
- Replication allows you to keep multiple Hazelcast IMDG clusters in sync by replicating their state over WAN environments such as the Internet. Hazelcast IMDG supports both Active-Passive and Active-Active modes to most common scenarios. The replication mode is to be chosen in unison with the replication strategy of the whole system.

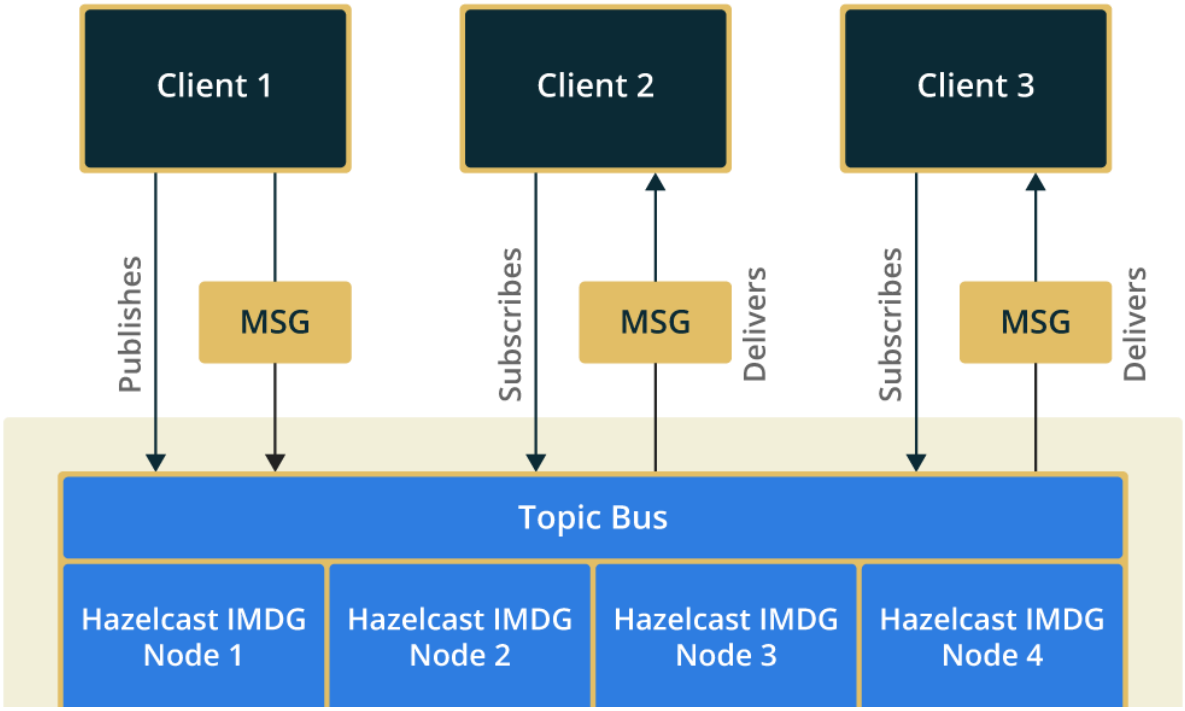


Figure 11. Hazelcast Client-Server Architecture

Hazelcast IMDG Distributed Topic Bus

5.9 MICROSERVICE COMPONENTS

Within the scope of KRIS5 the microservices are built on top of the following components:

- Hystrix
- Ribbon
- Feign

Spring Config client and Eureka Discovery Service Client are also compulsory components of the microservices but do not require a separate description within this chapter as they are already covered from their server side.

HYSTRIX

In order to add latency- & fault tolerance to a distributed system, Hystrix should be used throughout all services. In essence it is a library that helps control the interactions between distributed services and stop cascading failures and allows overwhelmed or failing services time to recover. The fallback can be another Hystrix protected call, static data, or a sensible empty value. Fallbacks may be chained so that the first fallback makes some other business call, which in turn falls back to static data.

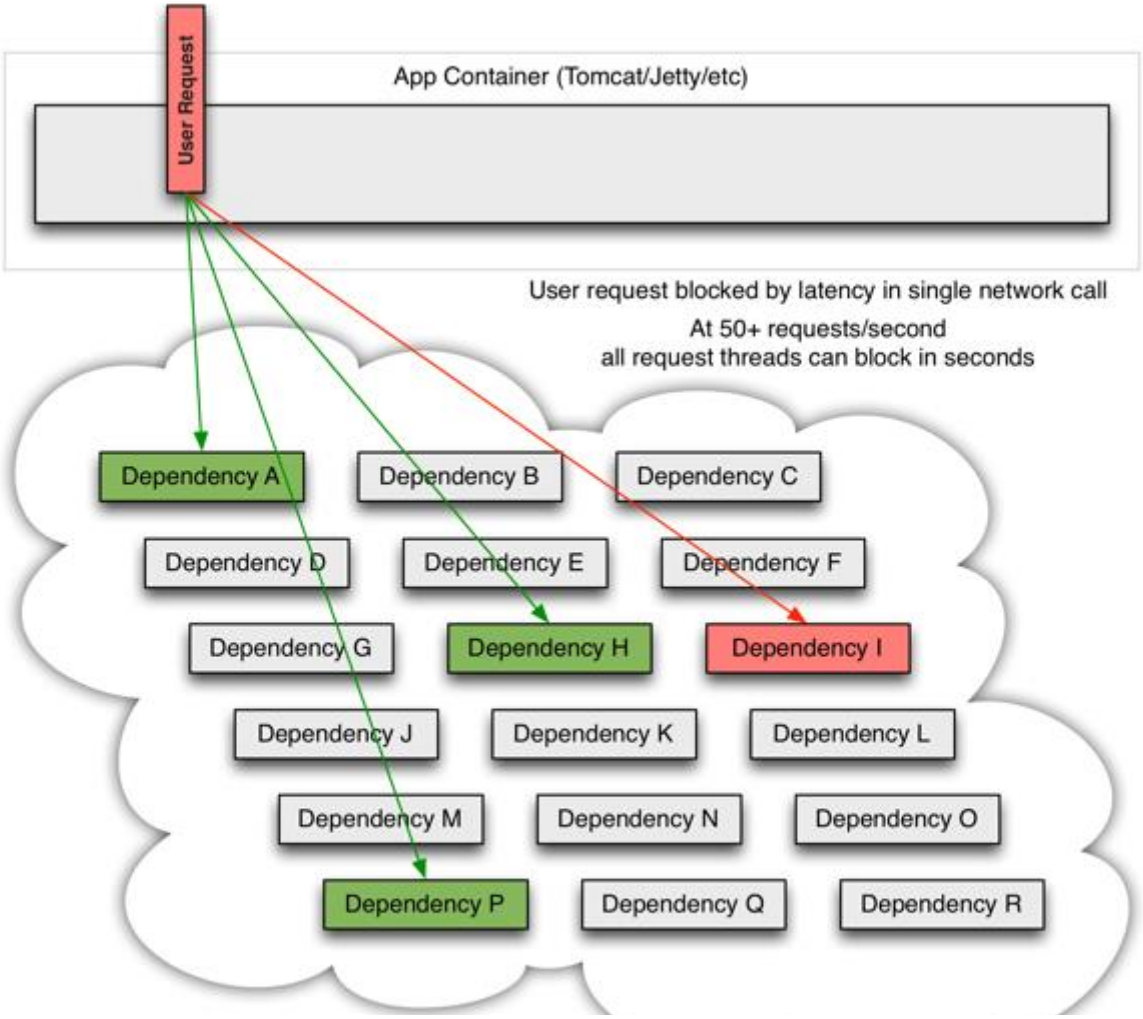


Figure 12. When one of many back-end systems becomes latent it can block the entire user request

Link (incl. image): https://github.com/Netflix/Hystrix/wiki

RIBBON

Ribbon is the client library used along with Eureka to make REST calls between services. It is built on top of the Apache HTTP client library. Ribbon provides client side load balancing between mid-tier services. It works in tandem with the Eureka client to determine the service instances available to receive an HTTP/HTTPS. The default load balancing algorithm is a simple round robin that can be customized if desired. See Figure 13 for a graphical representation on Ribbon usage.

Link: https://github.com/Netflix/ribbon/wiki

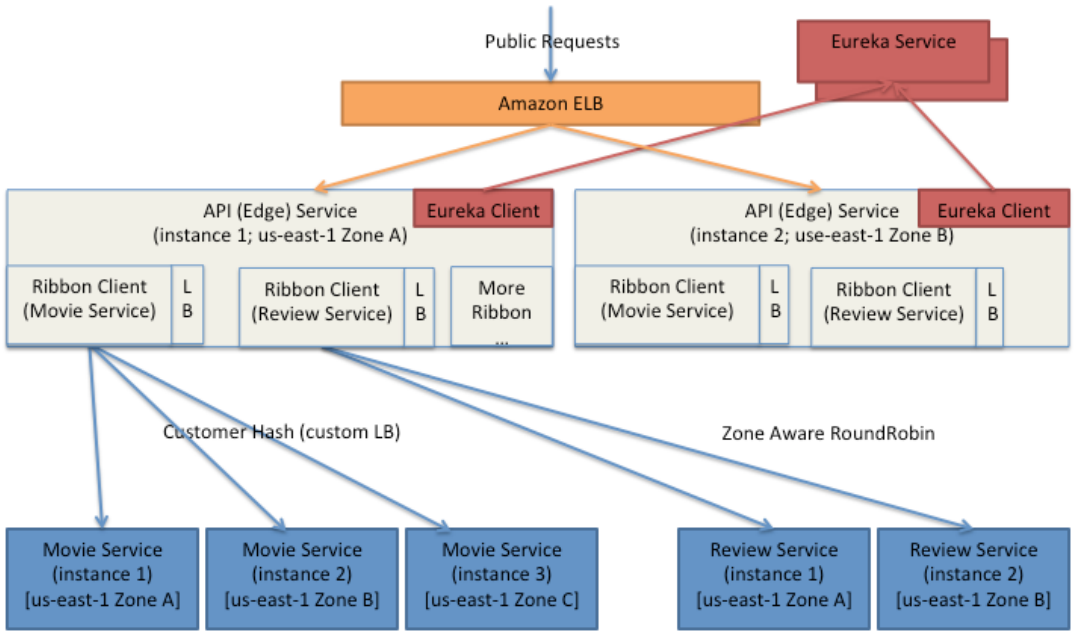


Figure 13. Typical (representative) multi-region, multi-zone deployment architecture at Netflix

FEIGN

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it, the actual implementation will be provisioned at runtime. Feign supports pluggable annotations, encoders and decoders. Spring Cloud used in KRIS5 adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

Link: https://github.com/OpenFeign/feign

DOCUMENT NR: 20180412-1
VERSION: 0.9
UPDATED: 2018-06-08
AUTHOR: ANDRES AAVIK
DOCUMENT TYPE: TECHNICAL
SECURITY: CONFIDENTIAL
BASED ON TEMPLATE NR 20120612-1, VERSIOON 1.0

5.10 OBJECT STORAGE

The OpenStack Object Store (files, data blobs etc.) project, known as Swift, offers cloud storage software so that you can store and retrieve lots of data with a simple API. It's built for scale and optimized for durability, availability, and concurrency across the entire data set. Swift is ideal for storing unstructured data that can grow without bound.

Link: https://wiki.openstack.org/wiki/Swift

5.11DATABASE

Any database types (relational, non-relational, in-memory, noSQL etc.) can be used based on the requirements of the microservice. The database design considerations are also described in chapter 3.1 “Microservice design”. It is however highly recommended to use PostgreSQL where possible. Within the context of KRIS5 for example all of the registry data should be stored using PostgreSQL.

DATABASE REPLICATION

PostgreSQL supports streaming replication. Streaming replication allows a standby server to stay more up-to-date than is possible with file-based log shipping. The standby connects to the primary, which streams WAL records to the standby as they're generated, without waiting for the WAL file to be filled.

Streaming replication is asynchronous by default (see Cascade replication), in which case there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby. This delay is however much smaller than with file-based log shipping, typically under one second assuming the standby is powerful enough to keep up with the load.[31]

There are two main PostgreSQL replication methods that should be considered depending on functionality of the microservice using the data store: cascade (asynchronous) and synchronous.

Cascade replication

Cascade replication is asynchronous and allows a standby server to accept replication connections and stream WAL (Write Ahead Log) records to other standbys, acting as a relay. This can be used to reduce the number of direct connections to the master and also to minimize inter-site bandwidth overheads. A standby acting as both a receiver and a sender is known as a cascading standby. Standbys that are more directly connected to the master are known as upstream servers, while those standby servers further away are downstream servers. Cascading replication does not place limits on the number or arrangement of downstream servers, though each standby connects to only one upstream server which eventually links to a single master/primary server.

[31] https://www.postgresql.org/docs/current/static/warm-standby.html

A cascading standby sends not only WAL records received from the master but also those restored from the archive. So even if the replication connection in some upstream connection is terminated, streaming replication continues downstream for as long as new WAL records are available. See Figure 15.

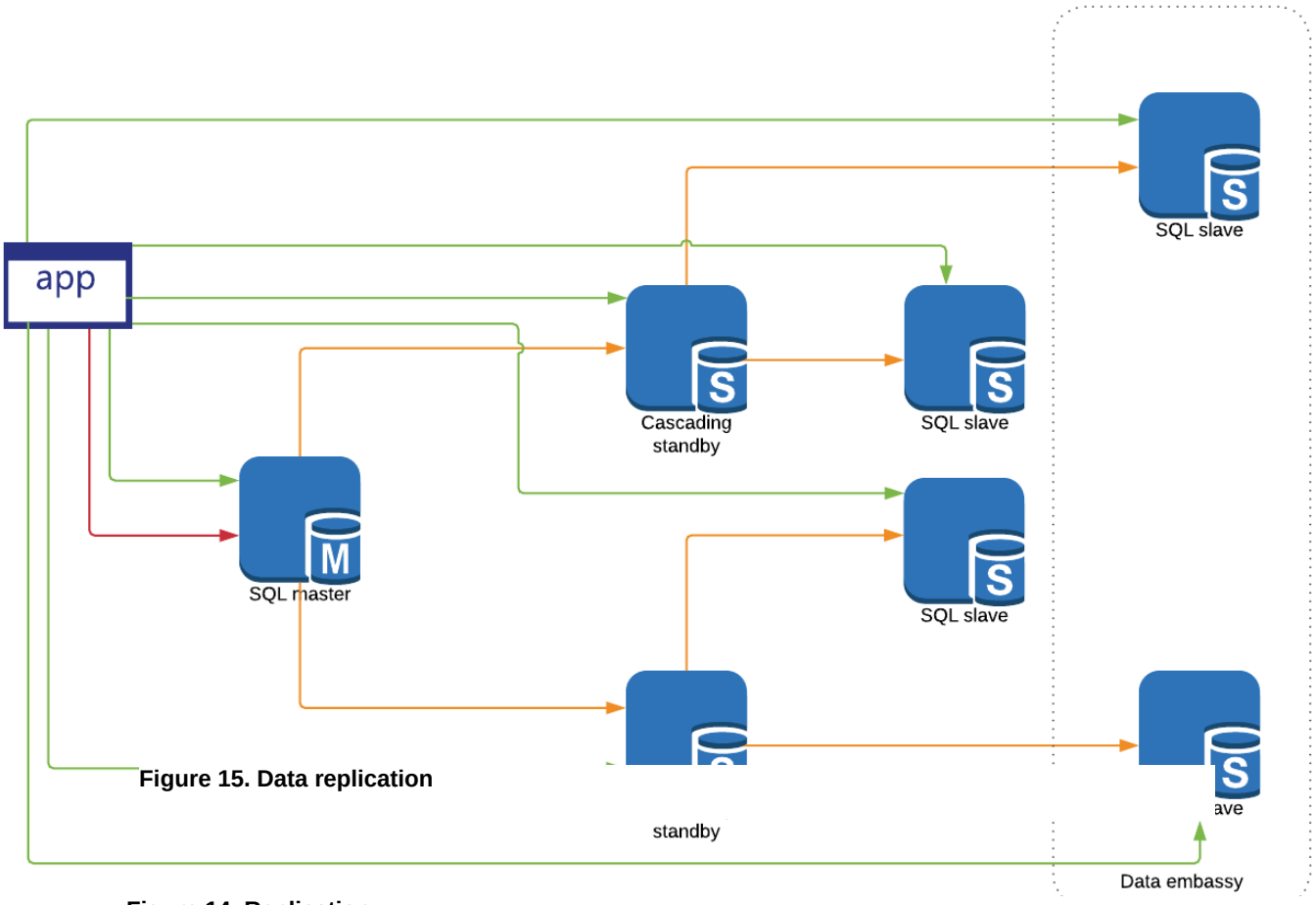


Figure 15. Data replication

Figure 14. Replication

Link: https://www.postgresql.org/docs/10/static/warm-standby.html#CASCADING-REPLICATION

Synchronous Replication

When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the write-ahead log on disk of both the primary and standby server. The only possibility that data can be lost is if both the primary and the standby suffer crashes at the same time. This can provide a much higher level of durability, though only if the sysadmin is cautious about the placement and management of the two servers. Waiting for confirmation increases the user's confidence that the changes will not be lost in the event of server crashes

but it also necessarily increases the response time for the requesting transaction. The minimum wait time is the round-trip time between primary to standby.

Link: https://www.postgresql.org/docs/10/static/warm-standby.html#SYNCHRONOUS-REPLICATION

In summary choose:

- cascade replication for maximum performance and minimal performance hit from the replication process itself
- synchronous replication if performance loss from the replication process itself is acceptable and absolutely no data loss is acceptable

OTHER TOOLS AND METHODS

Continuous Archiving and Point-In-Time-Recovery (PITR)

Continuous archiving and PITR should be set up for all PostgreSQL databases containing registry information.

The log records every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by “replaying” the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the file system backup and then replay from the backed-up WAL files to bring the system to a current state. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we do not need a file system snapshot capability, just tar or a similar archiving tool.
- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
- It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports point-in-time recovery: it is possible to restore the database to its state at any time since your base backup was taken.
- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a warm standby system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

Link: https://www.postgresql.org/docs/10/static/continuous-archiving.html

pg_rewind is a tool for synchronizing a PostgreSQL cluster with another copy of the same cluster, after the clusters' timelines have diverged. A typical scenario is to bring an old master server back online after failover, as a standby that follows the new master. The result is equivalent to replacing the target data directory with the source one. All files are copied, including configuration files. The advantage of pg_rewind over taking a new base backup, or tools like rsync, is that pg_rewind does not require reading through all unchanged files in the cluster. That makes it a lot faster when the database is large and only a small portion of it differs between the clusters.

Link: https://www.postgresql.org/docs/10/static/app-pgrewind.html

6 Scaling and backup

6.1 SCALING

CONTEXT - THE SCALABILITY CUBE

“The Art of Scalability” by Martin L. Abbott and Michael T. Fisher describes scalability using a cube model. The "Scale Cube" is composed of an X-axis, Y-axis, and Z-axis. The traditional method of scaling by running multiple copies of an application load-balanced across servers is the X-axis. /../ Decomposing an article into smaller services is the Y-axis. /.../ The X-axis is traditional load-balance scaling, and the Y-axis is embracing microservices. The Z-axis takes a similar approach to the X-axis—running identical copies of code across multiple servers. What makes Z-axis scaling unique is that it also borrows a page from the Y-axis, so each server is only responsible for a subset of the application rather than the application as a whole. /…/

The benefits of microservices notwithstanding, there's an added level of complexity when it comes to scalability. Instead of dealing with a single application running on a single server—or load-balanced across a few servers—you might have elements of an application written in different programming languages, loaded on different hardware, running on different virtualization hypervisors, and deployed across disparate cloud and on-premise locations. When demand increases for the app, all the underlying components have to be coordinated to scale, or you have to be able to identify which individual elements need to scale to address the surge in demand.

Using the Z-axis scaling approach from the Scale Cube allows you to segregate data across different servers based on routing criteria. You might route requests based on the primary key of the data being accessed, or based on customer type—sending paying or premium customers to servers with more bandwidth and capacity to deliver better performance.

With the legacy approach to IT infrastructure and app deployment the whole app had to be addressed as a monolithic entity. If demand spiked, the whole application had to be multiplied to accommodate the load, which meant multiplying the servers or virtual server instances on which the application was running.

[source: https://techbeacon.com/challenges-scaling-microservices]

LOAD-BALANCE SCALING

Any given microservice or object storage can be scaled in the traditional manner of replicating instances and sharing load among nodes.

VIRTUAL MACHINE / CLOUD SCALING

Given the granular setup of the architecture proposed within this document, it is easy and cost efficient to either manually or via the service provider’s API resize the virtual instance (eg CPU count, RAM amount or storage size). This approach can also be combined with the traditional load-balance scaling which will also have the much

desired and cost effective side effect of providing high availability per microservice. Also to be noted – if more than one node is used, scaling of the instances can be performed without any downtime as traffic will be automatically redirected to the other node.

6.2 ONLINE DATA EMBASSY PROBLEM STATEMENT

THE BASIC PROBLEM

How to, assuming imperfect communication, maintain and prove exactly one understanding of registry entries and of their order?

- The architectural design of KRIS5 has yet to solve this problem. It can be done at the system design level.

OTHER DESIRABLE PROPERTIES

Pre-emptive integrity checks. If an entry with an invalid signature exists in the registry, it assumes signature verification to discover. A self-healing system where violations of consensus for whatever reason can be automatically detected and corrected, would be beneficial

- Currently we are not aware of any existing solutions or architectural design concept to produce a solution that could „self-heal“. A solution could be developed as a separate project from KRIS5.

Active-active load balancing between sites. Since we already assume prevention of branches, it would be nice be able to operate the registry in an active-active regime perpetually

- The proposed architectural solution for KRIS5 supports active-active load balancing. The only negative aspect of running the proposed architecture in full active-active cluster where both identical system setups actively serve end customers is that synchronous database replication may be required for some databases serving specific microservices. In most cases there are options within the current architecture to direct customers towards the nodes in the same geographical location and thus avoid any data integrity issues (i.e. replication not yet done to other site).

Privacy of the data. If we are already employing some cryptography to distribute transactions, it would be nice for the transactions not to be readable without access to majority of nodes.

- There are several ways to interpret this specific

Active-Active cluster of KRIS5 architecture – identical setups in EST and Data Embassy can be seen on Figure 2.

PROPERTIES OF THE SOLUTION

Correct ordering of registry entries *must* be preserved

- By correctly applying the combination of asynchronous and synchronous database replication to the given microservices the registry entries will be preserved.

• It *must* be able to handle transaction loads of at least 1 entry per second
- Due to the scalable nature of microservice architecture proposed, loads of 1 entry per second are not an issue if all components are given enough resources.

• It *must* support more than two nodes of the consensus cluster
- All components within the proposed architecture can run in a clustered setup.

• A minority of “poisoned“ nodes *must not* be able to influence the state of other nodes
- If one node is „poisoned“ others will be affected. However with the proper WAL archiving setup, Point-In-Time-Recovery will at least mitigate the effects.

• Privacy preservation capabilities of the cluster nodes *can* be assumed
- Unknown what is meant here

• It *must* be possible to confirm successful formation of consensus on an entry
- Is possible within the current architecture concept. However the confirmation process cannot be solved with architectural design and should be solved by additional development (a highly specialized microservice to perform the verification).

7 Non-functional requirements

All the validation is done in Appendix 1 “rik-kris5-architecture-vision-appendix1-compliance-RIK nõuded arendustele v 6.0.xlsx”

1 – means that the requirement is fulfilled or is not in conflict with new architecture

0,5 – means it needs additional requirement change

0 – requirement is not valid

n/a – not applicable in this project scope or cannot be assessed.

? - need more information

7.1 NEW REQUIREMENTS

There are some new requirements to consider:

- To use git as code repository as configuration server needs to maintain files in git
- To use Docker or same kind of management for microservices
- To use Continuous Integration (CI) solutions in development is a must
- Data integrity verification should use data aggregation hashing
- Javascript and Typescript requirements like allowing it as a programming language, allowing versions of scripts used
- Javascript and Typescript package management requirements – what packages are allowed and which not (security analysis needed)
- Statecloud (Riigipilv) requirements, i.e. for scaling
- Data embassy requirements, i.e. fault tolerance, scaling, network connection security
- High availability requirements