Public Safety Secretariat of Mato Grosso Microservice Environment

João Paulo Preti
Federal Education Institute
Mato Grosso, Brazil
preti.joao@ifmt.edu.br

Adriano Souza
Federal Education Institute
Mato Grosso, Brazil
ah.driano@gmail.com

Tiago Lacerda
Federal Education Institute
Mato Grosso, Brazil
tiago.lacerda@ifmt.edu.br

Evandro Freiberger
Federal Education Institute
Mato Grosso, Brazil
evandro.freiberger@ifmt.edu.br

Abstract

This paper presents the microservice environment of the Public Safety Secretariat of Mato Grosso (SESP-MT) which was conceived to allow a migration process from SESP-MT monoliths and to absorb new organizational agile requirements. Despite the hype of microservice oriented architecture, it's an architectural style, with some general principles and as the nature of distributed systems, it can be organized in many different ways. As a result of this research, supported by IFMT, FAPEMAT and SESP-MT, 22 containers with several tools and services were assembled, tested and deployed in the SESP-MT environment integrated with the DevOps pipeline. Therefore the contribution lies in the successful environment implementation that can help other organizations.

1. Introduction

The Public Safety Secretariat of Mato Grosso (SESP-MT) is responsible for formulating, coordinating, executing and monitoring the State policy for the preservation of public order and safety in the State, which plans, coordinates and monitors several activities like ostensive police, investigative activities, fire fighting and prevention, weapons control, traffic safety, national security assistance, and partnerships with the Federal Government to enhance security in the State [1].

The wide variety of SESP-MT activities led to the development of big monolithic software applications that are not agile, take a long time to deploy software updates and that suffers from third software components that are incompatible with the legacy technologic stack.

With the aid of Mato Grosso State Research Support Foundation (FAPEMAT) and the Federal Institute of Mato Grosso (IFMT), a research project was carried out to deploy an environment that supports the development of microservices applications observing SESP-MT restrictions.

The project lasted 12 months and the activities were organized according to agile scrum methodology with sprints of 4 weeks duration. The first sprint was used to understand the organization of SESP-MT systems and the MoSCoW requirements prioritization technique was used for sprint planning. From the second sprint on there were monthly deliveries of executable products and a tested environment with a demo presentation followed by the next sprint planning. The activities addressed the following challenges: authentication and authorization, data repository, messaging, data log, monitoring and distributed transactions. The last sprint of the project was used for technology transfer with 7 workshops for the SESP-MT development team and for the teams of the two outsourced software houses.

Due to confidentiality agreements and security reasons the real applications and data schemas cannot be presented, but an hypothetical scenario and a functional prototype were developed to demonstrate and validate the microservice environment and serve as a microservice template as well.

Therefore the paper is organized as follows: section 2 presents the scenario utilized as a reference for the development of the functional prototype. Section 3 presents the project decisions for microservice base implementation. From section 4 to 8 are presented how the previous challenges were addressed. Finally, section 9 presents the full environment organization deployed in SESP-MT.

2. Scenario

This section describes the functional prototype business process scenario, a particular case of 190 service (e.g. 911 USA). Is a vehicle theft scenario, where the victims must contact 190, identifying themselves and informing them of the data related to the fact, such as location, description of the car and even the suspects. After that, the victim has to go to the nearest police station and finalize the police report. Figure 1 describes the process.

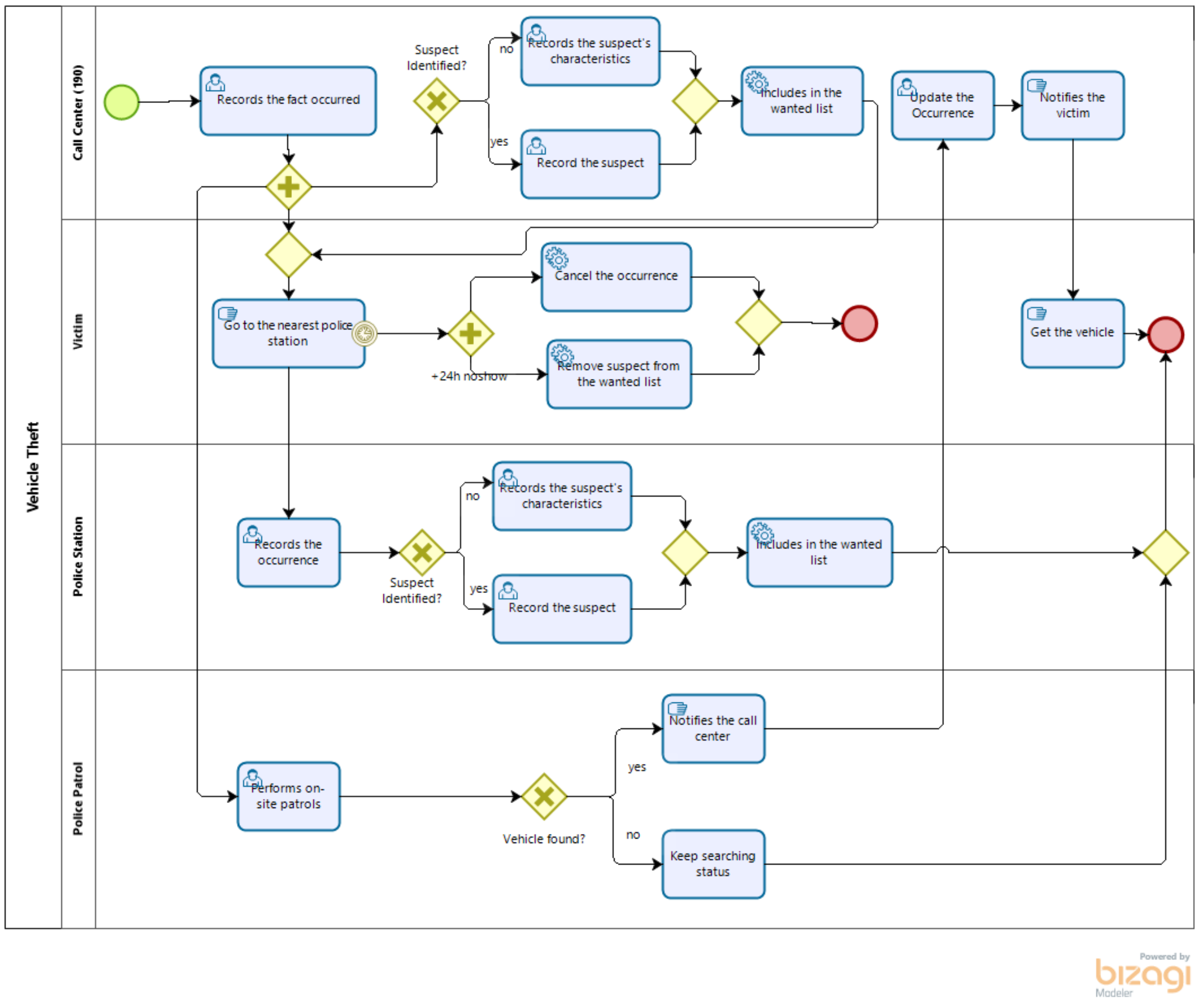

Figure 1. Functional prototype scenario for a vehicle theft.

In the scenario, the call center can record the suspect theft characteristics which is included in a wanted list that can be used by police patrol. The victim is oriented to address to the nearest police station in order to officialize the occurrence and to complement the record with more details. The absence of the victim in a police station in the next 24 hours implies the cancellation of the started workflow.

Meantime, the nearest police patrol is notified about the occurrence of the vehicle theft and directed to the theft location. Patrols can notify the call center which are responsible for notifying the victim.

In this particular scenario we extracted 5 domains:

- the **occurrence**, which represents the vehicle theft occurrence in the scenario;
- the **wanted list**, that contains suspects and people that are being in search by police;
- the **police patrol**, which represents police cars and its route plans;
- the **human resources**, that can represent police officers and administrative staff;
- a **global registry**, which contains citizens data and location addresses. This domain can be used by several other domains, like the occurrence domain to obtain address data of the location of the vehicle theft or data about the victim.

These domains led to the definition of five microservices which can establish a communication relationship as shown in Figure 2.

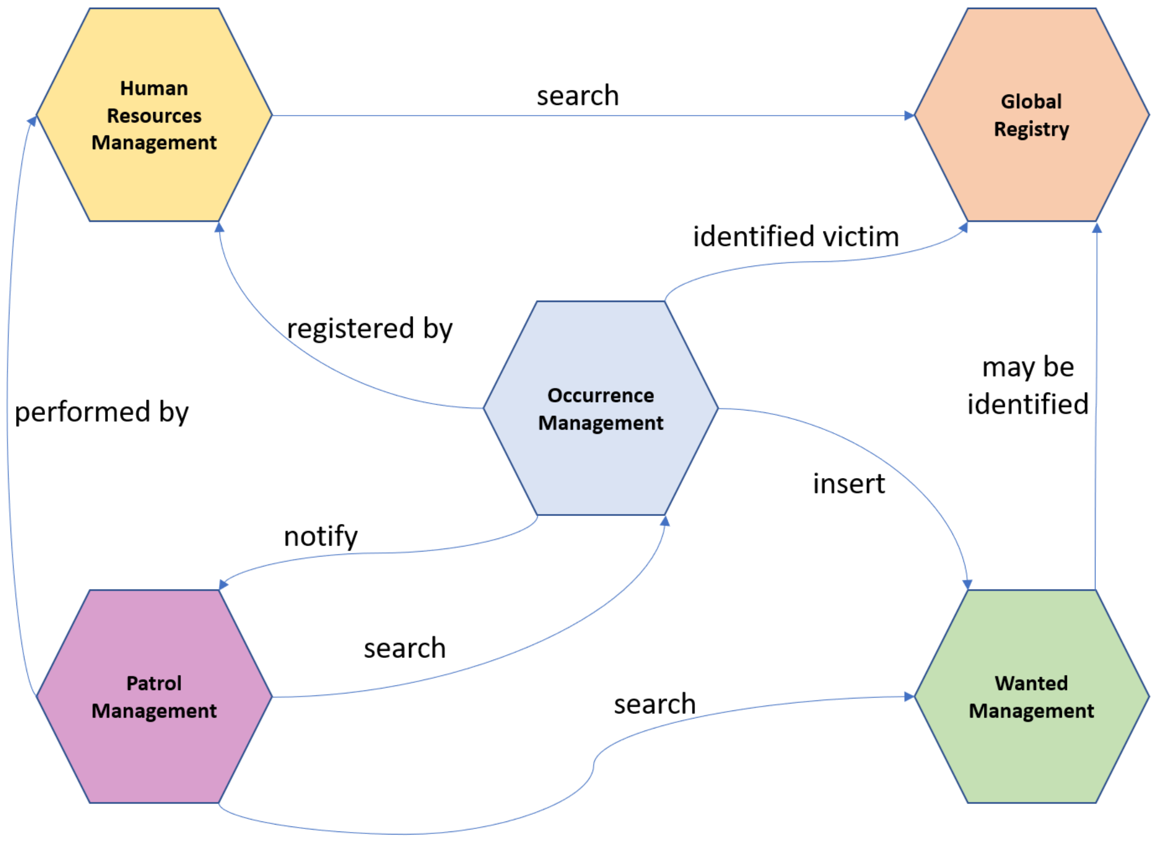

Figure 2. Five Microservices of the functional prototype.

3. Project

Each microservice is a spring boot project with endpoints mapped based on the REST Resource Naming Guide. All microservices are based on:

- Microservice domain: core of the microservice that represents entity classes and/or value objects;
- Service: contain the business logic;
- Controller: provides an access interface to the microservice, exposing its visible operations, as well as establishing a format for input and output data;
- JSON/HTTP: general communication protocol;
- JSON/AMQP: communication protocol based on message queues;
- RESP/TCP: redis serialization protocol for caching;
- GELF/UDP: graylog extended log format for data log;
- Database connector: for data persistence;
- Oauth: for authentication and authorization using JWT.

All data exchanged through microservices is in JSON format converted from a Data Transfer Object (DTO) [3]. The external exposure of DTO is in order to hide the domain model organization or complexity [4].

In view of the previous scenario, its tasks ramifications and business domains of the process, it is possible to reach the model represented by a class diagram in Figure 3, identifying the concepts that must be dealt with, presenting their relationships and respective minimum data [2].

4. Repository Model

In a microservice architecture there is the "Database Per Service" design pattern that establishes the policy of one service per database. This means that each microservice operates as an intermediary between the customer and their data and these can only be modified through the interface of that service [5].

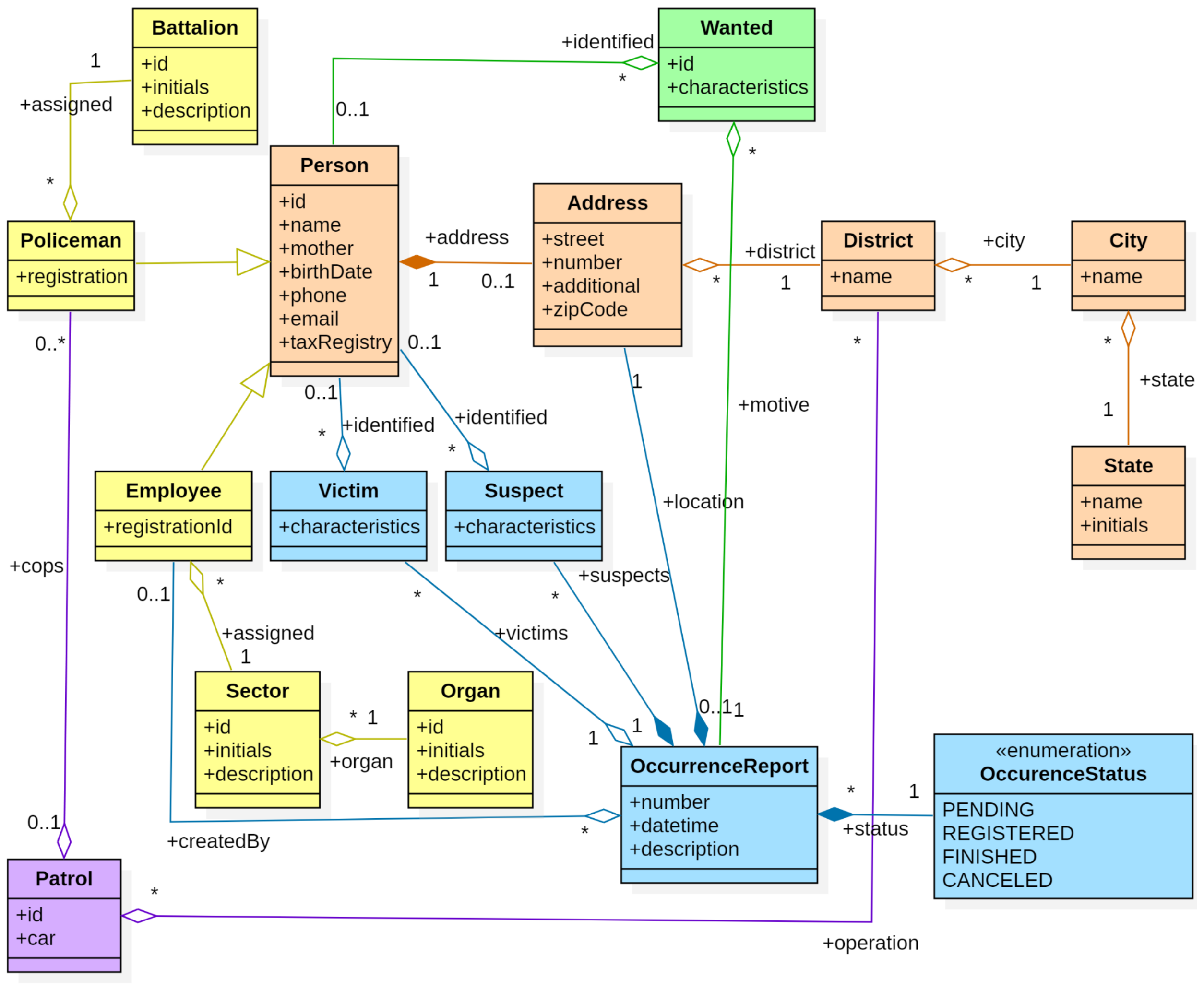

Figure 3. Global schema with different colors referencing five different microservices domains.

Regarding data access, it is important to highlight that SESP-MT pointed out the impossibility of immediately adopting a microservice architecture in which each microservice has its own database.

Thus, two strategies were proposed to be adopted for the microservices that share the same database [1]:

- **Permissive**: allow the visibility and reading of entities that are out of their context (just read access, never write access);
- **Restrictive**: deny the reading of entities that are out of their context, a request must be made to the responsible microservice.

The strategy choice is not mutually exclusive, but aiming to facilitate a future migration process to a microservice architecture, the restrictive strategy is more appropriate since it minimizes the dependence on the global database schema and facilitates the process of a physical separation [6]. The recommendation followed by the team is to always use the restrictive strategy first, but not limited to it.

In view of the distributed system characteristic and the granularity of microservices, the use of a client-server caching at the service level was configured to improve response time [7]. An investigation of client-server tools led to the choice of Redis as a cache product solution [1].

5. Authentication and Authorization

The authentication step consists of identifying users and systems, authorization, on the other hand, is a later step and is related to the privileges that are granted to a given user [8]. To provide these capabilities to the functional prototype architecture, three approaches and the characteristics of each, need to be considered.

In the **local authentication and authorization** approach, each microservice is responsible for both authentication and authorization and establishes a strong relationship with the concept that each microservice must be autonomous and independent [9][10]. Despite providing the most loosely coupled architecture, the probability that each microservice needs different authentication mechanisms is very low, which would result in the multiplicity of the same code in different microservice projects and would significantly increase the effort of corrective and evolutionary maintenance of this requirement in different projects.

On the opposite side, the **global authentication and authorization** is an all-or-nothing approach, that is, if the microservice is registered and the user is identified, then he is authorized [11][12]. As positive aspects we do not need to replicate the implementation of this non-functional requirement, so corrective and/or evolutionary maintenance becomes easier due to centralization. It also helps to keep focused on business rules. On the other hand the microservice has no control over what the user can or not do, that is, a finer granularity of permissions is not achieved, therefore unwanted occurrences should be investigated through log auditing, for instance. Also availability is impaired, considering that because it is centralized, a failure can cause disruption of the system as a whole.

We opted for an hybrid approach, **global authentication and authorization as part of the microservice**. As positive aspects there is the finer granularity of permission aspects and centralized management becomes easier, as it has fewer responsibilities. There is also no network latency regarding authorization aspects and authorization failures are limited to the microservice where the failure occurred. On the other hand there is more code for the developer, since he must implement these aspects for each microservice and a permission matrix is needed to understand the context of user permissions on each microservice.

Although the understanding that an authentication and authorization strategy implemented by each microservice is completely adherent to the weak coupling, autonomy and independence between microservices, it is extremely costly, bringing relevant negative aspects and little-justified practical benefits, being able, for example, to use redundancy to increase the availability of global authentication.

5.4. Identity and access management tools

Eight tools were considered: Keycloak, Identity Server, Gluu, CAS, OpenAM, Shibboleth IdP, LemonKDAP and Okta. The analysis observed seven characteristics: OpenID Connect and OAuth support, multi-factor authentication support, presence of an administrative user interface, identity brokering support, open source, commercial support and github stars.

Despite these tools being very similar, Okta is not open source, Shibboleth IdP does not have an administrative user interface and CAS, Shibboleth IdP and LemonLDAP have just commercial support by third party companies. For the remaining tools, the main characteristic which led to the choice of Keycloak was its support by the community which is the largest and most active.

6. Messaging

The purpose of messaging is to allow each microservice to handle, exchange messages and collaborate with other microservices to meet a customer's request [13]. This communication can be managed directly or through a broker/bus.

The advantage of using a broker/bus is the use of an advanced, customizable, language-independent set of features, maintained and tested by a community or organization with frequently updated versions providing greater security and robustness. The downside is that we have one more centralizing element in the architecture where a failure in the messaging service can compromise the availability of various services.

It is not within the scope of this paper to elucidate the differences between a message broker and a message bus, but to investigate two widely used products in the market and what led to the choice of one of them to compose the microservice environment [14].

Among the best known products on the market are RabbitMQ and Apache Kafka, although Redis can be used as a broker, its focus of use is caching, which is already used in the SESP-MT microservice environment for that end. Kafka and RabbitMQ are both open-source and support the publish-subscribe model.

RabbitMQ supports several protocols that are industry standard such as AMQP, MQTT, STOMP, XMPP and JSON-RPC, which allows its use in

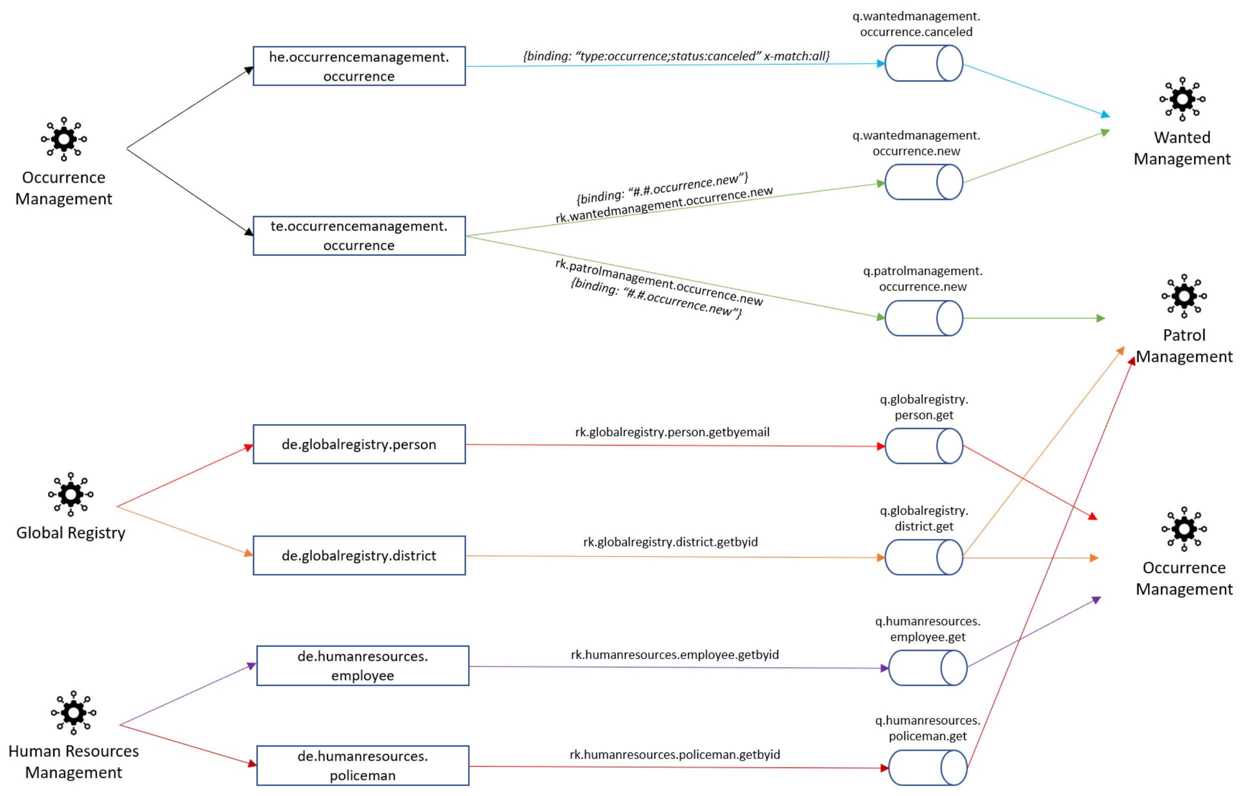
Occurrence Management
he.occurrencemanagement.occurrence
te.occurrencemanagement.occurrence
{binding: "type:occurrence;status:canceled" x-match:all}
{binding: "#.#.occurrence.new"}
rk.wantedmanagement.occurrence.new
rk.patrolmanagement.occurrence.new
{binding: "#.#.occurrence.new"}
q.wantedmanagement.occurrence.canceled
q.wantedmanagement.occurrence.new
q.patrolmanagement.occurrence.new
Wanted Management
Patrol Management
Global Registry
de.globalregistry.person
de.globalregistry.district
rk.globalregistry.person.getbyemail
rk.globalregistry.district.getbyid
q.globalregistry.person.get
q.globalregistry.district.get
Occurrence Management
Human Resources Management
de.humanresources.employee
de.humanresources.policeman
rk.humanresources.employee.getbyid
rk.humanresources.policeman.getbyid
q.humanresources.employee.get
q.humanresources.policeman.get

Figure 4. AMQP entities for the five microservices of the functional prototype.

different scenarios. The use of standardized protocols also allows the exchange of RabbitMQ for any other AMQP broker.

Kafka uses its own protocol over TCP/IP for communication between applications and the cluster, that is, Kafka cannot simply be exchanged as it is the only one to implement the protocol.

Considering that the architecture and environment established for communication between microservices is not intended for a specific product, but rather a flexible architecture and environment to support a gradual migration process of several SESP-MT software products, it is understood that RabbitMQ is a more suitable product for this purpose, as it supports several standardized protocols and supports a greater variety of communication models between microservices.

6.1. Implementation

In Figure 4 there is the representation of all AMQP entities (Exchanges, Queues, Bindings) present in the functional prototype.

There is no general convention applied to the naming of messaging resources. For this reason, each institution can adopt a standard that best suits the team's needs. Thus, the standard adopted in the implementation is a proposal that aims to facilitate the identification of resources by interpreting their name. This pattern divides the resource name into levels, separated by dots, as shown:

resource_type.publisher|consumer.entity[.action | type]*

We defined three minimum levels that generally apply to any scenario, but this study should also take into account the problem domain and should be suitable for new or migration projects.

Level 1: Identification of the resource type, which may contain one of the following values: **de** (direct exchange), **fe** (fanout exchange), **he** (header exchange), **te** (topic exchange), **q** (queue) and **rk** (routing key).

Level 2: Identification of the service responsible for the resource. We chose to use the publisher in the composition of the name.

Level 3: Identification of the entity referring to the resource.

Level 4: Optional, identifies the action related to the resource or a separation by type of instances of the entity. This level will be exclusively used by resources such as Queue and Routing Key.

Below is an example of a value for a routing key used in the functional prototype:

rk.occurrencemanagement.occurrence.new

It is noticed that the resource is a routing key (rk), whose microservice publisher is the Occurrence Management that will publish messages containing as payload an Occurrence whenever a new one is created.

7. Data Log

Logging is related to recording relevant events in a computer system. This record can be used to restore the original state of a system or for an administrator to know and analyze the application's historical behavior. A log file can also be used for auditing and diagnosing computer system problems.

As for the log related to a monolith and a microservice architecture there are several differences. The first difference is that in microservices there are more components, which ends up generating more logs since each microservice generates its own set of logs. The second, is that the logs of each microservice in isolation are not enough to understand the flow of events, since microservices are constantly communicating with others to achieve a goal. It is necessary to understand how microservices are interacting.

Another challenge is the scope and types of logs generated by each microservice, which tends not to be uniform, since the independent and distributed nature of this architecture is not only a technological solution, but also a scope and teams.

7.1. Good Practices

We enumerate good practices that help effectively address these logging challenges in a microservices architecture [16].

Aggregate and analyze log data at a **centralized point** for a holistic view of the application.

Use personalized and **unique identifiers** to contextualize and map interactions between microservices.

Write **custom filters** for the central log as the structure can vary from one microservice to another.

Persist log records. Microservices often run inside infrastructure such as a container - which lacks persistent storage. A basic and essential best practice in this case is to ensure that the log data is written somewhere it will be persistently stored and will remain available if the container is shut down.

It is possible to achieve persistence by modifying your container's source code or settings to ensure that logs are written to an external storage volume. An easier approach, however, is to run a logger that will collect data from the containerized microservice in real time and aggregate it into a trusted storage location.

It is interesting to **keep all the logging code** in the application source code in production. Thus, it is possible to create a strategy to enable a greater or lesser detail of events, without the need to restart the application.

If an ERROR occurs, **collect and record as much information as possible**. This action can take time but is not critical as normal processing has failed, rather than waiting for the same condition to occur.

Allow the application to change the **logging level dynamically** without the need for a reboot.

Remember that logs are often used for monitoring, identifying and troubleshooting issues.

7.2. Data Log Attributes

As for the pertinent information that is registered in the log, we list: microservice ID, user ID, IP address, request ID, correlation ID, instant of time (UTC), method/operation name, running stack trace.

The HTTP protocol allows the use of Request-ID and X-Correlation-ID headers. The request identification allows identifying all the events occurred by that request, whereas the correlation identifier is very important in distributed systems as it allows tracking the flow of events by task (which can involve several requests). Without the use of this type of information, we could filter the logs by user and track the timestamp to identify the order of the execution, but this would not be enough to identify when the log of a specific activity triggered by this user starts and ends.

As for the level of logging enabled in the SESP-MT environments, these were defined as follow:

- TRACE/DEBUG: Level utilized only in development environments and therefore disposable. The purpose is to assist the developer in debugging the service;
- INFO: The test environment is intended to help system users who are testing/validating the services that will be made available. Considering the reduced scope of users, usage

time, data and even performance, the INFO level is sufficient for application validation.
- WARN: Level utilized in production environment, since the log is only used to identify problems or check if there is a shortage of computational resources.

7.3. Technical Solution

It is important to mention that a test was carried out with Spring Boot Admin as a possible decentralized strategy for monitoring log events, but it proved to be unfeasible considering that the tool does not allow a holistic view of events, only an isolated view per microservice, not allowing the follow-up of the communication flow.

An alternative for consolidating and centralizing log messages is the use of its own messaging broker (RabbitMQ), creating its own queue for receiving log messages. This type of solution can be interesting for performing customized actions and also allows for a simple centralization solution of log messages in distributed systems, but does not provide any additional resources for storage, filtering and analysis, being in charge of its own implementation and/or use of third/additional software. One must also take into account the overhead that this type of solution can cause to the broker, requiring a resizing of computational resources for messaging.

The test and final choice was based on Elasticsearch and Logstash tools. These are two very popular open-source tools for large data storage, processing and log data collection, respectively.

Logstash is used to aggregate data log from all microservices simultaneously and transform that data before it is indexed in Elasticsearch, which can run complex queries to retrieve complex summaries of the data.

8. Monitoring

Monitoring is a process of capturing the behavior of a system through health checks and metrics over time. This helps detect anomalies: when the system is unavailable, has an unusual load, exhausts certain resources, or otherwise does not behave within its normal (expected) parameters. Monitoring involves collecting and storing long-term metrics, which is important, more than anomaly detection, but also root cause analysis, trend detection and capacity planning [17].

In this project, the focus is on monitoring microservices, the message broker and data log [18]. Monitoring of bare metal resources, virtual machines or containers is not in the scope, this monitoring is in charge of the tools used by the operation team.

We envision three elements for monitoring the functional prototype environment:

1. **Microservice**: observe each microservice and its execution status;
2. **Message Broker**: observe the AMQP entities and the Erlang VM itself;
3. **Data log**: observe the historical behavior of the microservices as a whole and use this data to identify problems or possible events that may be compromising the use of the system.

Monitoring **microservices** is a necessary task in order to ensure that systems behave as intended. More than just checking its status at the moment, that is, whether it is active or not, metrics such as memory usage, storage, input/output rates, logs and requests contribute to the identification of performance issues that can impact the application and cause its fault. Since microservices are Spring Boot projects, the use of Spring Boot Actuator was a natural choice, which is a subproject of Spring Boot that includes features that help monitor and manage Spring applications. It is possible to obtain metrics, application health data and auditing in a simple way, either through HTTP requests or with the help of JMX.

For the **message broker**, RabbitMQ allows the monitoring of messaging broker metrics and system metrics through a graphical management interface and HTTP API that exposes a series of RabbitMQ metrics for nodes, connections, queues, message rates and so on. This is a convenient option for development and in environments where external monitoring is difficult or impossible to introduce. However, RabbitMQ's management UI has limitations, the monitoring system is interconnected with the system being monitored, it only stores recent data and generates a certain amount of overhead.

Long-term metric storage and visualization services such as Prometheus and Grafana are more suitable options for systems in production, since it decouples the monitoring system from the system being monitored, allow a long-term metric storage, access to other related metrics such as Erlang runtime and a a more powerful and customizable user interface.

Another positive aspect is that Prometheus can collect metrics from microservices, serving as a centralizing and persistent point of metrics for various elements of the architecture.

As for visualization tools, two are very popular: Grafana and Kibana, but since SESP-MT already knows and uses Grafana, this was the selected tool.

Data log lacks a visualization tool that eases the construction of filters and log visualization. Two tools are very popular in this scenario: Graylog and Kibana.

The Graylog tool was selected to compose the microservice environment because it is specialized in logs, has a lower learning curve and makes the alert system available free of charge.

9. Complete Environment

The complete environment is composed of 22 containers, as can be seen in Figure 5.

In this figure the containers are separated in four different layers. Containers in one layer may directly connect to containers in the same level and in the layer right below. The containers were named according to the applications inside them.

The first layer has a single container, the Theft Management. This is the frontend of the solution. It is a SPA that was built using Angular framework. This application makes direct requests to all the services in the second layer and provides a graphical user interface to the solution.

The second layer is represented by the business logic containers. There are six containers inside this layer, five of them are related to the theft management business itself and the last one - the authentication service - holds the authentication logic.

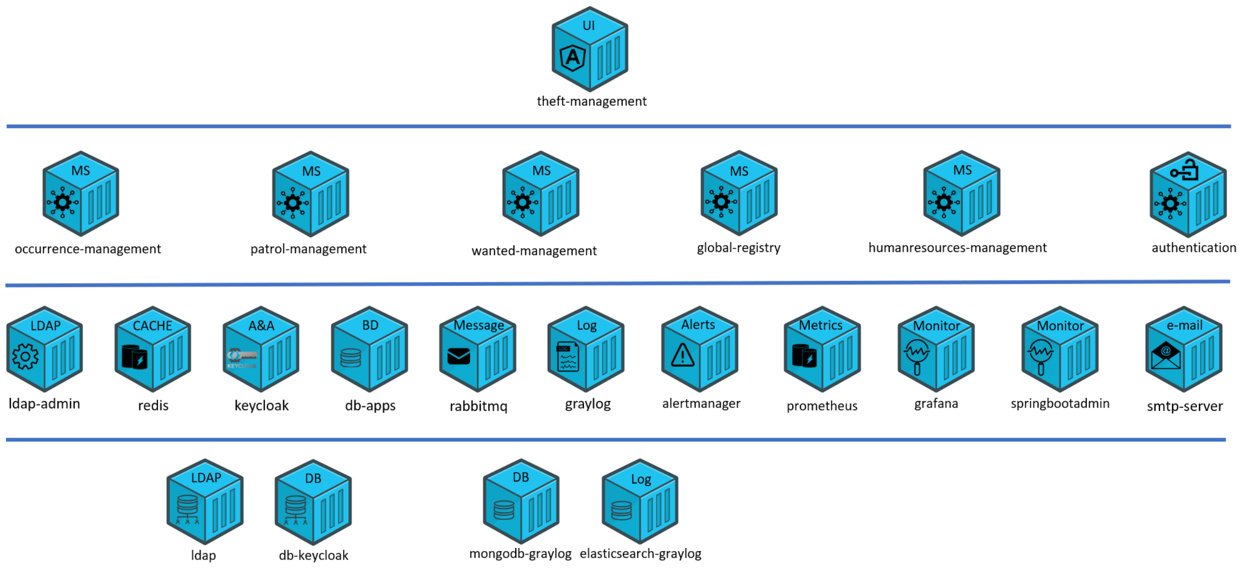

Figure 5. 22 Containers of the microservice environment.

The authentication service was created to isolate and centralize the authentication logic regarding the Keycloak connection and the social authentication. This service also provided an interface to the authentication endpoints for external applications, which are those that are not deployed on the same network as Keycloak.

The third layer is the biggest one. There are eleven containers related to the tools used by the business logic services. Tool, in this context, can be understood as a software that was not built from scratch, but is part of the overall solution, being an existing software that was configured according to the needs of this solution. This layer is composed by the following tools:

1. **LDAP Admin:** an interface for the LDAP Service in the fourth layer. This tool is used to manage the users, groups and roles used by Keycloak;
2. **Redis:** an in-memory data structure store. Redis is used by the business logic services to store the cache data;
3. **Keycloak:** an identity and access management solution used to manage the authentication and authorization to the business

logic services using both the LDAP Data and the authentication data inside the DB Apps database;

4. **DB Apps:** a PostgreSQL database used by the business logic services and Keycloak;
5. **RabbitMQ:** an AMQP broker used for message exchange between the business logic services;
6. **Graylog:** a log management solution used to centralize the logs produced by the business logic services;
7. **Alert Manager:** an alert tool used by Prometheus and Graylog to manage the alerts regarding infrastructure warnings;
8. **Prometheus:** a monitoring tool that serves as an interface for the metrics and alert data to be used by Alert Manager and Grafana;
9. **Grafana:** a monitoring tool used to create dashboards. Uses data from Prometheus to generate graphs regarding the resources consumption and state of the services. Also uses the SMTP Server to send email alerts when necessary;
10. **Spring Boot Admin:** a monitoring tool that provides a basic dashboard with the business logic services state;
11. **SMTP Server:** a SMTP solution used to provide the capability to send emails for the Alert Manager and Grafana.

The fourth and last layer is represented by the storage containers. These containers were necessary because some of the tools in the third layer needed to store data. There are four containers in this layer represented by both relational and non-relational databases.

In the functional prototype the container environment was initially orchestrated using Docker Compose, but currently at SESP-MT these containers are deployed in a kubernetes cluster managed by the Rancher tool and the continuous integration pipeline is based on GitLab CI/CD.

10. References

[1] Preti, J.P., A. Souza, E. Freiberger, and T. Lacerda, "Monolithic to Microservices Migration Strategy in Public Safety Secretariat of Mato Grosso", ICECCE 2021, (2021).

[2] Khononov, V., Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy, O'Reilly Media, S.l., 2021.

[3] Martin, R.C., Clean Code: A Handbook of Agile Software Craftsmanship, 2008.

[4] Newman, S., Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015.

[5] Khononov, V., Balancing Coupling in Software Design: Successful Software Architecture in General and Distributed Systems, Addison-Wesley Professional, S.l., 2021.

[6] Publications, Z., Migrating to Microservice Databases: Good database design principles explained, O'Reilly Media, 2017.

[7] Pacheco, V.F., Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices, Packt Publishing, 2018.

[8] Bánáti, A., E. Kail, K. Karóczkai, and M. Kozlovszky, "Authentication and authorization orchestrator for microservice-based software architectures", 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), (2018), 1180–1184.

[9] Gutierrez, F., "Security with Spring Boot", In F. Gutierrez, ed., Pro Spring Boot. Apress, Berkeley, CA, 2016, 177–209.

[10] Dikanski, A., R. Steinegger, and S. Abeck, "Identification and Implementation of Authentication and Authorization Patterns in the Spring Security Framework", pp. 7.

[11] ShuLin, Y., and H. JiePing, "Research on Unified Authentication and Authorization in Microservice Architecture", 2020 IEEE 20th International Conference on Communication Technology (ICCT), (2020), 1169–1173.

[12] Bánáti, A., E. Kail, K. Karóczkai, and M. Kozlovszky, "Authentication and authorization orchestrator for microservice-based software architectures", 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), (2018), 1180–1184.

[13] Bakshi, K., "Microservices-based software architecture and approaches", 2017 IEEE Aerospace Conference, (2017), 1–8.

[14] T, S., and S.N. K, "A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming", arXiv:1912.03715 [cs], 2019.

[15] Ionescu, V.M., "The analysis of the performance of RabbitMQ and ActiveMQ", 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), (2015), 132–137.

[16] Kim, T., S. Kim, S. Park, and Y. Park, "Automatic recommendation to appropriate log levels", Software: Practice and Experience 50(3), 2020, pp. 189–209.

[17] Jiang, Y., N. Zhang, and Z. Ren, "Research on Intelligent Monitoring Scheme for Microservice Application Systems", 2020 International Conference on Intelligent Transportation, Big Data Smart City (ICITBS), (2020), 791–794.

[18] Cinque, M., R. Della Corte, and A. Pecchia, "Microservices Monitoring with Event Logs and Black Box Execution Tracing", IEEE Transactions on Services Computing, 2019, pp. 1–1.