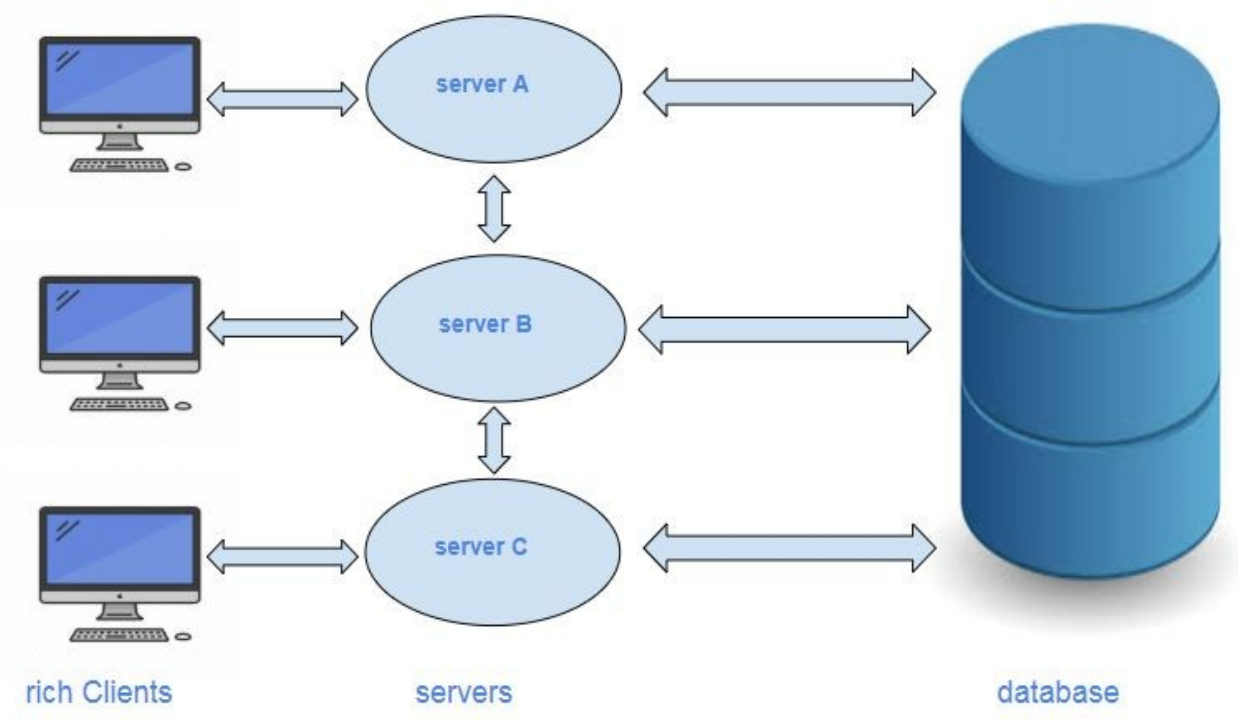
eNanoServices/embedded nanoservices architecture.

1. Introduction

This paper describes a general view of the eNanoServices architecture.

2. Before micro-services

Before micro-services became popular, applications for a typical big or medium-size company might look like this.



In this example, we have three applications running in their own servers, A, B and C. They would be supported by different teams and have their own UI, normally involving a rich client. They would talk to each other using http or a message bus, such as Tibco. These applications might be

independent, but they could become tied by the monolithic data base on the right. And eventually they will.

This approach also means that applications would have to be deployed at the same time, typically during a quarter release. If you've ever been on a production deployment conference call with 200 people late at night on a Saturday, you already know this is not the best approach.

Typically, the servers would keep track of the user's session by using cookies and storing the session in memory or the database, combined with some kind of cache.

Scalability, redundancy, failover are not built into this architecure and would need to be provided using ad hoc solutions.

Since these applications would have their own rich clients, deploying them also means deploying the corresponding rich clients to the computers of the internal users. The users, which might number in the thousands, will have their own setup and this might create troubles and require technical support.

User Interfaces have been migrating to the browser, because it provides a standarized interface, they are deployed from the server and through the same interface internal and external users can be supported. This migration is not easy and, for sure, not cheap because...

- rich clients tend to run quite a bit of business code, for performance reasons, which will need to be moved to the server.

- rich clients, typically, do not enforce a separation of the UI from the rest of the code by using, for example, the MVC design pattern. This results in the UI code being entangled with the business code.

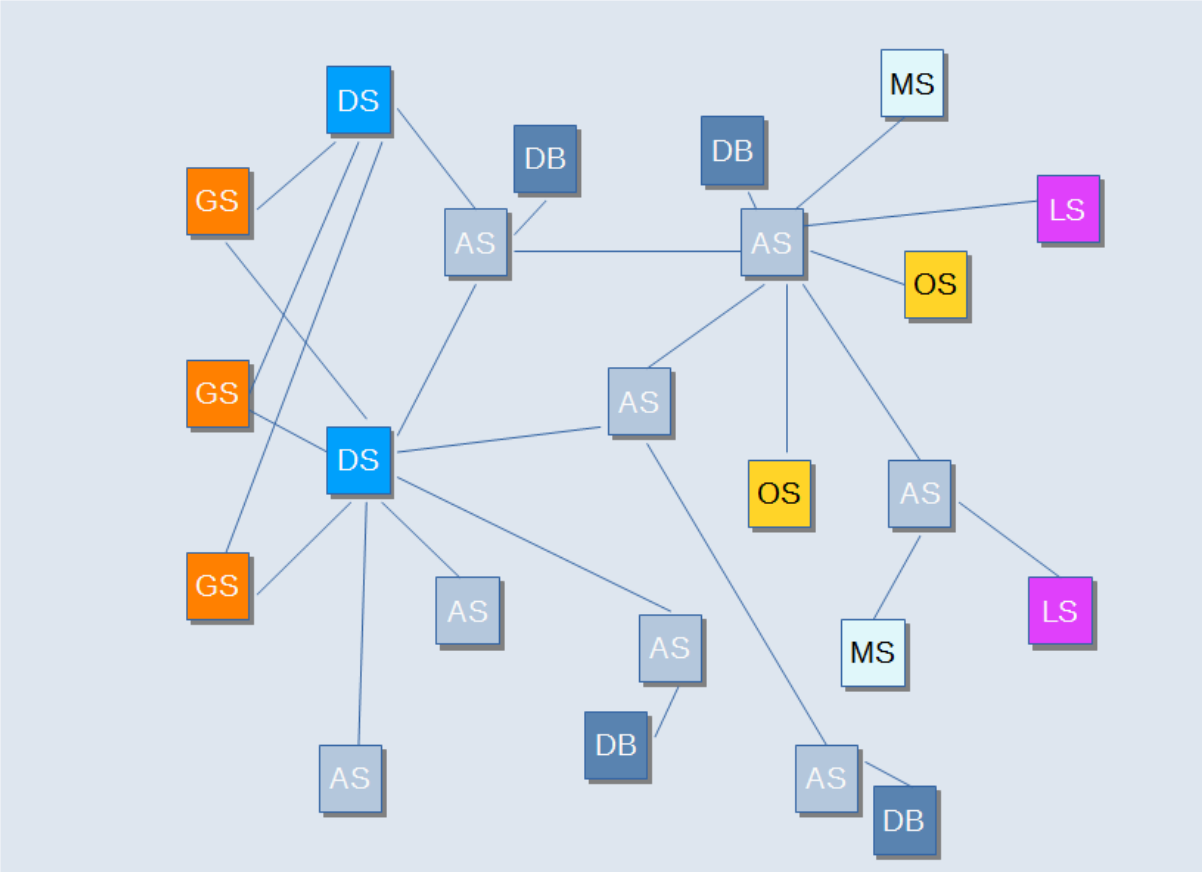
3. Micro-services architecture

Micro-services aim at solving the problems with monolithic applications. They became popular with the advent of the cloud.

A typical microservices architecture is organized around individual web services, which tend to have these characteristics:

- micro-services should be as independent as possible
- micro-services boundaries tend to reflect business units boundaries
- they should be supported by small teams
- they should be independently deployeable
- they should be scalable, redundant and impervious to server or communications failures

This image shows several types of micro-services running in their own servers.



A big company might have hundreds of them interacting with each other. We can see that some of them are application services, implementing business code, while others are auxilary services, required by the architecture

These are some examples of auxiliary services needed for support, besides the databases:

- GS – Gateway Services accept user's requests and direct them to other services.

- OS – Operation Services keep track of request/responses involving more than one server, such as database transactions.

- LS – Logger Services provide logging for other servers, normally for interactions involving more than one service. Individual services might also implement their own logging.

- MS – Metric Services allows administrators to keep track of the health and performance of the various services. They'll be interested in throughput, response times, latencies and resources.

- DS – Discovery Services are used to identify the server or servers implementing a specific endpoint (URL).

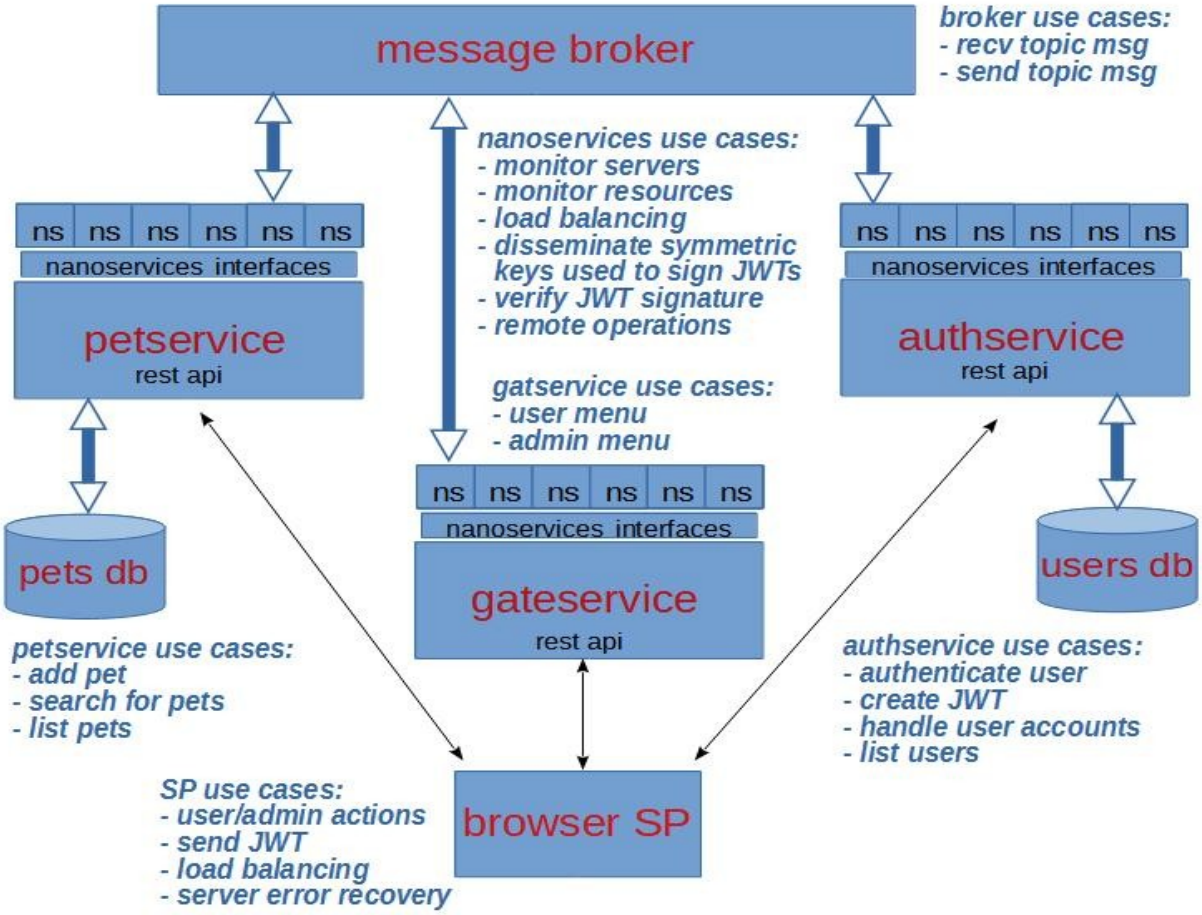
This architecture provides many advantages, such as independent deployment of services, redundancy, fault isolation and scalability.

It has its own disavantages: testing and development become more complex. Identifying a problem might involve tracing the requests/responses across multiple services by looking through the logs.

But its main disavantages are: communications become more complex and chatty and the administration of all those servers becomes harder. In other words, application code becomes easier but the problems are pushed to the network.

4. eNanoServices architecture

Let's take a look at a general view of the architecture we are using along with the "Pets" web application we've developed to illustrate this approach.



We refer to the auxilary services, at least some of them, as nanoservices and we are co-locating them with the application code. We still need some auxiliary services, such as a gateservice, which is the entry point for the application and it has a role supporting host redundancy.

Some nanoservices will be exposed to the application code by a well-defined interface.

This approach reduces the number of servers and traffic and simplifies development, deployment and support.

These are some of the details:

- Each microservice has its own database to decouple it from the rest.

- Internal communications between servers involve a message broker running on its own server.

- We are using JWT for web security. In our example, authservice creates the JWT [token] after the user logs-in. This token is sent to the client which sends it to the corresponding server with every request. Servers don't keep or share user's sessions, so this is a stateless approach.

- We rely on static configuration as little as possible. Within a host we obtain and update the servers' configuration at run-time.

- We deal with server faults and redundancy. We assume that a server can go down at any time and the requests need to be routed around the failed server.

- We support remote operations, mostly for database operations involving more than one service. At this point we are not supporting database transactions within the framework: if needed, the application code needs to implement them.

- Micro-services can become tied by the user interface, specially if implemented as SPAs (Single Page Application). SPAs are more responsive, since only part of the web page has to be updated, but they could tie the micro-services. The UI is part of the application code which needs to address this issue.

- We support servers running different micro-service versions and different nanoservices versions.

- Although not available at the time of this writing, service redundancy will be extended to other hosts.

.

It’ll be more clear after we provide host redundancy, but this approach implies a hierarchical organization, which allow us to reduce the communications bandwith and simplify development and monitoring.