Architecture of a software service or resource oriented

Olivier Sallou [IRISA] - JDEV 2017

Overview

API

Background tasks

Business processes

Scalability/Resilience

Persistent storage

Monitoring

API

Presentation vs functional

Web UI vs web server

- Separation of Web interface, interact with web server using a public API
    - Customer can create their own portal or scripts interacting with your web server, web ui is only an additional component
    - Modifications of web ui do not impact server code

API, API, API....

Keep it stable:
- add fields and endpoints, never remove
- Keep fields optional, limit mandatory ones

Document it (swagger, autogenerated from code => readthedocs)

Use api keys to track usage, limit pure anonymous access

Anonymous vs authenticated access, public vs private

Monitor its usage, latency…

Public vs private api

Use HTTP Authentication headers with tokens, avoid cookies (impacts scalability)

Answer fast ! Avoid blocking answers (long answer, file transfer, etc.)

Good practices

For endpoints returning data (index search, list of data, …):
Expect pagination (from, limit , …)

Nice to have: add pagination links in your answer with urls to use to go to next or previous page, example:

```
{
data: [ …],
previous: { url: ‘https://…./page/10},
next: { url: ‘https://…./page/12},
current: { url: ‘https://…./page/11}
}
```

Use standard HTTP answer codes (404, 401, 403, 503…)

Use REST endpoints

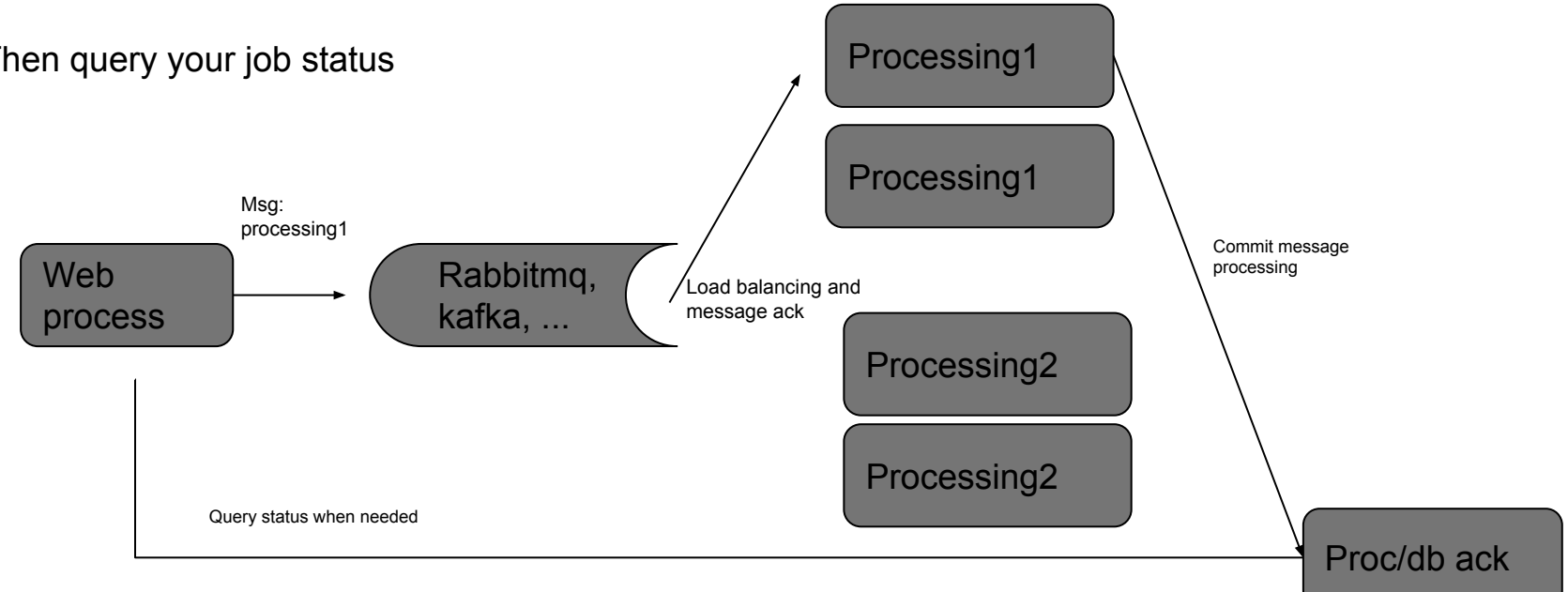
Use HTTPS with a real certificate (from domain,Let’s encrypt, …)

Long running tasks

Background tasks

Send to messaging process and scale to background processes
If process fails, message resent to another process
If many tasks, will be queued

Then query your job status



Business/Functional processes

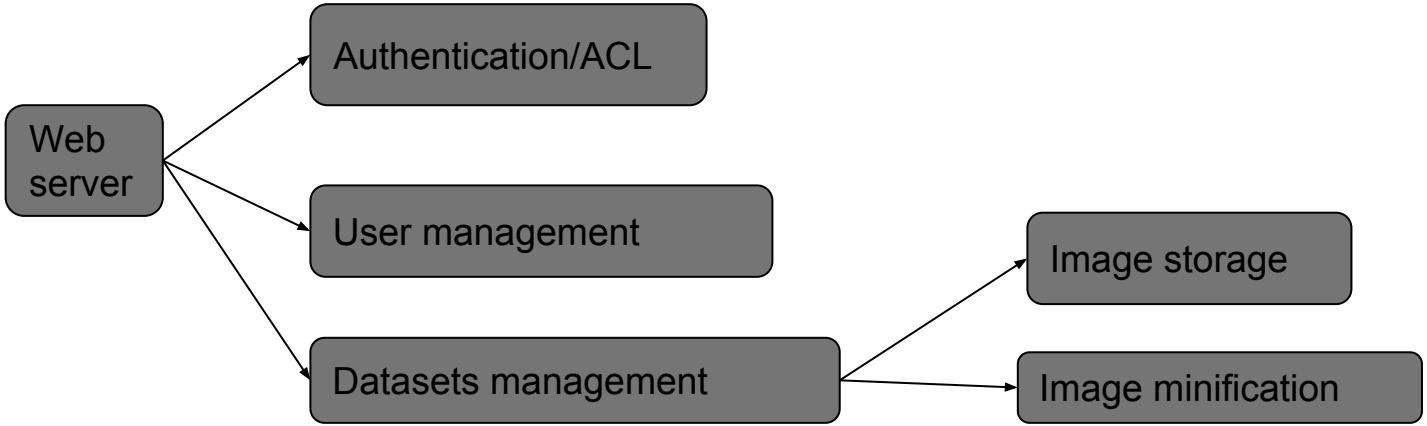
1 process = 1 business task

Split your monolith in micro/macro services

1 service = 1 function (authentication, ACL, image processing, …)

Ease upgrades, maintenance, evolution and scalability: impacts only the service/process

Services talk via API using same rules as web API via a private proxy/messaging

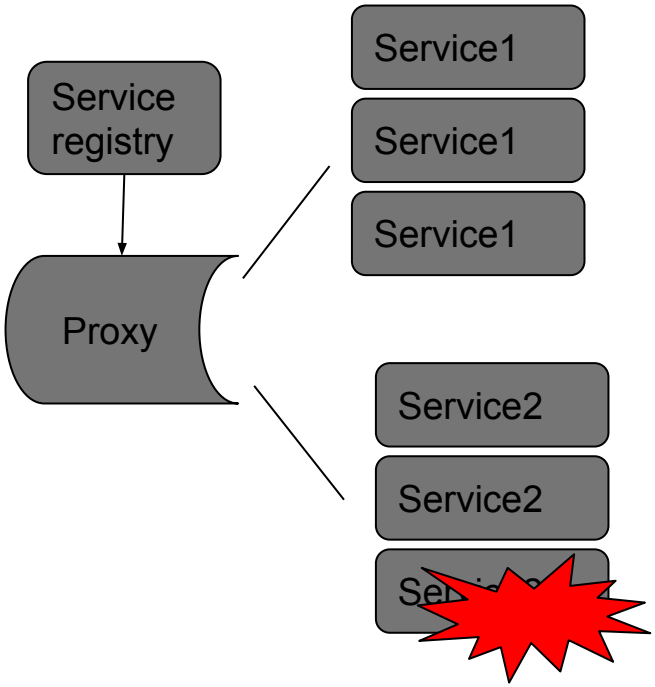


Adds complexity in management

Involves more monitoring

A good idea is to support distributed tracing on global or per request mode to see what is going on…. (zipkin for example)

Scalability
&
service discovery

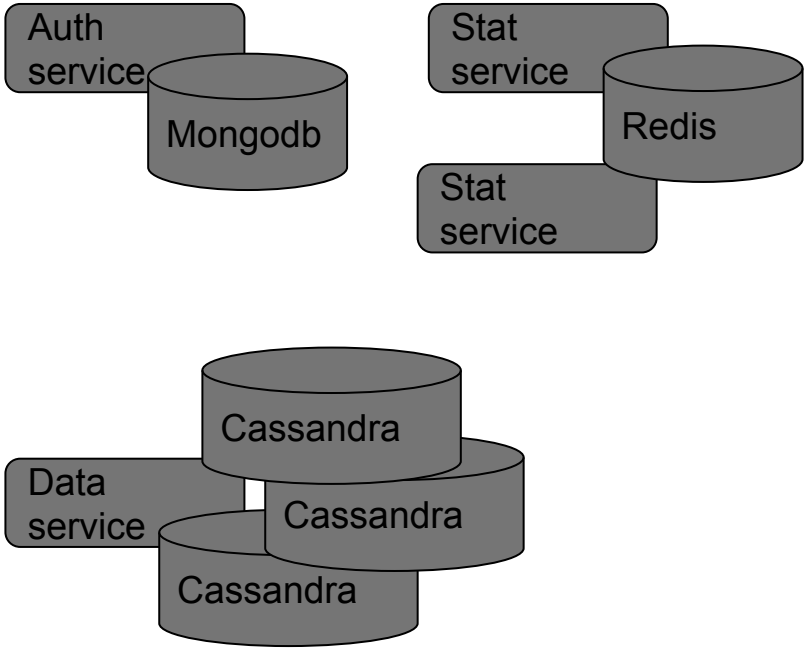


Use dynamic proxy (nginx, haproxy)

Service discovery + service checks (consul, ...)

Databases

Use the best for your needs



Do not stick to 1 database

Use the best database for your service (relational, document based, graph, indexation => mysql, mongodb, redis, cassandra, elasticsearch…), vs your needs (scalability, answer time, security, backups …)

Persistent storage

How do you use/scale your data?

File system , database, object storage, ….

Think about future needs for scalability, will be hard to modify afterwards...

Does service needs direct file access or only a storage for remote download

Manage data lifetime:

- when and how a data/user should be removed?
- Update your indexes (elastic, ...) when adding but also removing data….

Take care batch upload/removal with database indexes… index updates are costly and will impact database performance

Backups

Backup your data

Dump your databases

Data should be backup too or using reliable systems…… ;-)

Monitoring

Moniting and alerting

Get usage/latency stats (with tools like prometheus)

Alert on failure or unexpected behavior (nagios, prometheus-alert, …)

Centralize your logs (elk, graylog , …) : at scale you'll have many services!
Add alert rules in your log

Support log rotation!

In your code, differentiate errors with service impact vs errors impacting 1 user request (database connection failed vs wrong parameter in an API call)

Authentication

Do you need to manage users?

If you don’t need… use external authentication systems (oauth with google, shibboleth, ...). Just get their email/id from external identity providers

Limit user management if you can to avoid managing passwords, account validity etc.

User management = high security rules !

Use tokens, not cookies

JWT tokens are secured, encrypted tokens that can be sent over http or other protocols and can include the information needed to identify the user.

On authentication, generate a token and send it back to your web UI. UI will add this token to the next requests. Token can be decoded on any server managing the request without the need of shared storage/system (for cookies).

It can then be transmitted to other services to track user info while keeping it secret during message transmission.

Deploy/Ship your service

Containers are a good idea ;-)

Containers are an easy way to ship your service to other users. Will contain ready-to-go environment

Will also ease the deploy/updates in your dev/prod environements

Tools: docker, docker-compose, kubernetes, rkt, lxd …

Run anywhere

You may start on a local server, then switch to public/private clouds.

Think about it from the beginning when designing your application and selecting its core components (storage, network accessibility, …)

Thank you!

Twitter: @osallou

GitHub: https://github.com/osallou

Bitbucket: https://bitbucket.org/osallou/