Development view

2024-10-08

**Last modified** : 2024-10-08

**Last global review** : *<perform regular global reviews of the view at least once a year while the project is active and mention the date here>*

**Document status** : *<'WIP', 'DRAFT', …​>*

# Introduction

This view describes the development aspects of the application. It describes the code to be produced and how to write it properly.

The other views of the document are accessible [from here](./README.adoc).

The project glossary is available [here](glossary.adoc). We will not redefine the functional or technical terms used here.

## Reference Documentation

Mention here the reference (defined at a IS level) architecture documents. This file should never summarize their content under penalty of quickly becoming obsolete and unmaintainable.

Development documentary references

| N ° | Version | Document title / URL | Detail |
| --- | --- | --- | --- |
| *1* | *1.0* | [*https://www.w3.org/WAI/fundamentals/accessibility-intro/*](https://www.w3.org/WAI/fundamentals/accessibility-intro/) | *WAI* |

# Not ruled

## Points subject to further study

Points subject to further study

| ID | Detail | Status | Subject holder | Deadline |
| --- | --- | --- | --- | --- |
| *ED1* | *The choice of Angular or React.JS for the frontend is still under study. This does not impact the back part of REST services.* | *WIP* | *Dev team* | *Before 2040* |

## Assumptions

Assumptions

| ID | Detail |
| --- | --- |
| *HD1* | *Even if this point is not yet fully validated, the application will require a JRE 9 + to take advantage of the Java libraries and frameworks essential to the project.* |

# Constraints

List here the constraints related to the software architecture, this includes for example but not only:

* The obligation to use a framework or a specific technology
* Maximum licenses or development budgets
* Tooling (IDE, …​)
* Continuous integration
* The applicable code quality standards and thresholds
* Test methods and objectives (coverage rate, breakdown by type of test, etc.)

Example 1: Code coverage should be at least 60%

Example 2: The module should be based on the JPA framework for persistence

Example 3: The application will be built, tested and deployed continuously with each push via the Gitlab-CI platform

# Non-functional requirements

Contrary to the constraints which fixed the framework to which any application had to conform, the non-functional requirements were given by the project decision-makers. Schedule interviews to determine them. If some requirements are not realistic, mention this in the reference points to be decided.

## Accessibility

Should this application be accessible to the blind/visually impaired? deaf person?

If so, what level of accessibility? Preferably refer to the Web Content Accessibility Guidelines ([WCAG 2.0](https://www.w3.org/TR/WCAG20/)).

There are other accessibility standards. Be careful to correctly assess the target level (neither over-quality nor under-quality):

## Ergonomics

### Ergonomic charter

In general, we refer here to the body’s ergonomic charter. List any specificities. Do not duplicate here the accessibility constraints listed above.

### Specificities on widgets

Very precise ergonomic behavior can have a fairly strong impact on the architecture and impose a library of graphic components or another. It is strongly advised not to customize existing libraries (high maintenance cost, high complexity).

Example 1: The tables must be sortable according to several columns.

Example 2: Most screens will be fitted with accordions.

### Fonts

Describe here the fonts to use for web pages, applications, or generated documents.

The choice of fonts follows licensing constraints. In order to ensure legal security aspects of the project, pay attention to commercial policies subject to royalties (in particular policies of Microsoft such as Times New Roman, Courier, Verdana, Arial) and which do not allow to produce documents without going through their editors (Word, …​).

Redhat, for instance, provides four families of TrueType fonts (Liberation) under an Open Source license that is legally secure and compatible with the Monotype, Courier-New, Arial and Times New Roman.

### Responsive website

List the multi-media display constraints. When possible, use modern frameworks (such as AngularJS or React.js). There are several levels of adaptation of web pages:

* Static (fixed page width).
* Dynamic (automatic resizing, sizes are expressed in %).
* Adaptive (distances are expressed in units, the size of which depends on the support).
* Responsive (the content and its layout depend on the medium).

A responsive design comes with its constraints (code duplication, increase in the volume of data to be downloaded by the client, complexity, more end-to-end tests to be expected…).

### Progressive Web Apps (PWA)

Specify whether the application is progressive. PWA applications are HTML5 web applications having all the attributes of native applications (offline-first, fast, adaptive, accessible from the OS, …​)

Example: Application X will be fully PWA. Tests will have to demonstrate that the site will continue to operate with the network shutdown and that the pages load in less than 5 seconds using 4G Mobile Internet access.

### Supported browsers

Specify which browsers are supported if your project comes with a Web GUI.

When addressing an audience whose you do not manage browsers (such as a website on the Internet), the best option to make things intelligible and clarify the issues is to negotiate with the stakeholders of the project a minimum percentage audience supported based on <https://gs.statcounter.com/ouvernstatistics>]. For example: "Support 95% of browsers".

Supporting old browsers (IE in particular) can generate prohibitive additional costs and security issues. In all cases, the additional costs of testing on multiple platforms should be assessed. There are good (paid) tools like Litmus or EmailOnAcid to render websites and HTML emails on a combination of OS / player type (PC / tablet / mobile) / browser very large (in the order of 50). This type of site is essential for a general public application.

Example 1: The intranet application X must work on internally qualified browsers (see [Ref xyz])

Example 2: Application Y being an internet application targeting the widest possible audience, including terminals in developing countries. It will have to support Firefox 3+, IE 8+, Opera 6+.

Example 3: Application Z is aimed at the broadest audience and with reasonably old systems and will therefore have to support: Firefox 6+, Chrome 8+, Opera 8+, IE 10, Edge.

### Internationalization (i18n)

Specify the constraints of the application in terms of i18n: location of labels, direction of the text, adaptable layout, specific color code, date format, currencies, display of decimal separators, etc.

Example 1: The GUI X will be translated into 25 languages ​​including some Asian languages ​​and Arabic.

Example 2: Date formats and other input fields must be perfectly localized for maximum user convenience.

### Offline mode

Specify whether the application must be able to continue to function without Internet or LAN access (very common for applications used by professionals on the move for example).

These can be classic heavy clients (Java, C, …​) having their local database that can be synchronized back to the office. They can also be PWA applications (see above) using a service worker for static resources and browser storage (local storage, IndexedDB database).

Example 1: The application will be developed in Java Swing with local storage based on an H2 database synchronized with the common database by REST calls.

Example 2: The mobile application will be in PWA mode, entirely written in HTML5 with local storage to store the day’s data into the browser.

## SEO Requirements

SEO (Search engine optimization) concerns the visibility of a website through search engines (like Google or Baidu).

Example 1: No indexing necessary or desired (Intranet site).

Example 2: The static pages of the site must follow good SEO practices to optimize its visibility.

## Ecodesign requirements

Ecodesign consists of limiting the environmental impact of the software and hardware used by the application. The requirements in this area are generally expressed in WH or CO2 equivalent.

List the eco-design requirements relating to software here.

Example: The cumulative emissions of Service A should not exceed 100KgCO2/year.

# Target architecture

## Software stack

### Stack profile

Detail the technologies chosen from the organization’s catalog. If there are any discrepancies with the catalog, specify and justify it.

Example: this application has "Spring Web Application" profile with exceptional use of the JasperReport library.

Example: Using Reacts.js on an experimental basis within the organization. Validated in architecture committee on …​

### Software stack

List here for each component the main libraries and frameworks used as well as their version. Do not list the libraries supplied to the runtime by the application servers or the frameworks. There is no need to list every minor library: focus on the structuring software components.

Example:

Software stack example

| Library | Role | Version |
| --- | --- | --- |
| *Angular2 Framework* | *GUI JS Framework* | *2.1.1* |
| *JasperReport* | *Document generator of invoices in PDF format* | *6.3.0* |

## Performance

Requirements are listed in the [Sizing section](./view-sizing.adoc#requirements).

Even though performance campaigns are planned, experience shows that most of performance problems could have been detected early during development. It is therefore important that developers profile their code on their own workstation. This has to be set in the Definition Of Done of the project. It will not be possible to detect all the problems (scalability, concurrency, robustness, cache tuning, …​) but most of the response time or concurrency issues. There are many ways to simulate concurrency and load. You’ll find bellow some basic means accessible to any developer.

Backend side:

* Make sure that the server paging goes from the service call until the database (use FETCH FIRST x ROWS ONLY and [not LIMIT and OFFSET](https://www.postgresqltutorial.com/postgresql-fetch/)).
* Do not put in place unnecessary constraints in the database.
* In cases of very large volumes (from hundreds of millions), use database table partitioning.
* Don’t forget to add all the necessary indexes, use the analysis of the execution plan to verify the absence of full scans.
* Beware of SQL functions that 'break' indexes (like UPPER()) or use function indexes. Give priority to processing on the backend code side if possible.
* Activate the query logs (Hibernate example: org.hibernate.SQL = DEBUG,-Dhibernate.generate\_statistics = true) and check the SQL queries and their number (to detect in particular the very common [SELECT N + 1](https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping) issue).
* Have a minimum data set (more than one hundred records) even on a workstation.
* Check with a profiler (like VisualVM in Java) the memory consumption to detect leaks or over-consumption.
* Detect threads leaks or deadlock by counting the number of active threads over a significant duration (one full night for instance).
* Stress the API *a minima* (with injectors like JMeter or K6) using a progressive ramp.
* Track IOs (millions of times slower than memory accesses).

Frontend side:

* Limit the complexity of CSS (selectors or functions in particular)
* Use a profiler (like the one in Chrome)
* Favor asynchronous calls

Frontend and backend:

* Any resource (chain size, number of calls over a period, …​) must always be limited to a threshold (no "open bar" behaviors).
* Check that the size of HTTP requests remains below a few tens of KiB (excluding GET on files). Use "Sorting and Pagination, client and server pagination".
* Track network chatter: group requests when possible (you have to find a compromise with the previous rule). Use the SOLID (Interfaces Segregation) rule 'I'.
* Provide multi-valued endpoints (example: GET /people?List=id1,id2,…​) to retrieve several elements at once (must result in a single SELECT WHERE .. IN in the final query, not a loop in the code!)

Do not fall into the premature optimization trap : it is "the source of all the problems" according to Donald Knuth. Write the simplest code possible and follow a good design, only optimize it afterwards. Only optimize if it is worth it (Pareto law). Start with the most significant optimizations and do not waste time grabbing microseconds or even nanoseconds.

## Software factory specifics

Specify shortly whether this project requires a particular CI/CD configuration.

Example: Gitlab jobs will produce the software as Docker containers if all UT pass. The integration tests will then be run against the container. If all integration and BDD tests pass, the Docker image is released into Nexus.

## Blueprints

List applicable blueprints. Avoid own-in-house ones but give priority to wildly recognized ones from community, hence already proved and known to developers. Ideally, this should contain only links to external documentations.

Example : Follow [this tutorial](https://www.restapitutorial.com/) when designing a RestFul API.

## Code quality standards and automatic review

Make explicit the rules and the level of required code quality

Example 1: The quality rules to be used will follow the [SonarQube for Java rules](https://rules.sonarsource.com/java).

Example 2: The required quality level corresponds to the recommended [Quality Gate SonarQube](https://docs.sonarqube.org/6.7/QualityGates.html):

* 80% minimum code coverage
* 3% max of duplicate lines
* Level A in Maintenabily, Relability and Security

Example 3: Which language should be used for the code? Business terms in German (it is imperative to use business terms as recommended by the DDD) and English for generic technical terms.

## Notable patterns

Specify whether this project has implemented structuring patterns (GoF, JEE or other). No need to use patterns already supported by languages ​​or application servers (for example, IoC with CDI in a JEE 6 server).

Example 1: to deal with the combinatorial explosion of possible contracts and to avoid multiplying the levels of inheritance, we will massively use the decorator pattern[GoF], of which here is an example of use: <provide a diagram>.

## Specificities of the tests

Is there a particular methodology or technology involved in this project? What is the testing strategy?

Example 1: this project will be additionally covered by BDD (Behavioral Driven Development) acceptance tests written with Spock framework.

Example 2: this project will be developed in TDD (test first)

Example 3: Types of tests:

Types of tests

| Type of test | Time to invest | Manual or automated? | Type of module targeted | Target Coverage Rate | Detail |
| --- | --- | --- | --- | --- | --- |
| **UT (Unit Tests)** | Very high | Automated | Backend and Frontend | approx. 80% | BDD format: behavior specifications for classes and methods |
| **Executable specifications** | Very high | Automated | API | approx. 100% for the domain classes | Use mocks and doubles |
| **Contract tests** | Medium | Automated | UI/API links | approx. 100% of the calling code on the UI side and Spring controllers on the API side | Tests non-regression of exchanges when calling REST API operations (CDC = Consumer-Driven Contract principle) via the Pact and pact-react-consumer tools. |
| **Architecture tests** | Very low | Automated | API and batches | N/A, 100% of the code is validated by the tool | In particular, these easy-to-write tests will verify compliance with the rules of the hexagonal architecture. Use of the ArchUnit test framework. |
| **IT (Integration Tests)** | Low | Automated | Components calling external systems (databases, API …​) | 50 to 60% | Only test one external system at a time |
| **E2E (End-to-End testing)** | Low | Automated | UI | 30%, nominal cases (happy path) | Written in CodeceptJS, Selenium or similar technology. They will be limited to a role of smoke tests (detection of gross problems). These tests will not be mocked but will be run against an end-to-end instantiated linking chain. To avoid unnecessary work, these tests will be done at the level of entire features, not necessarily at each sprint. These tests will also serve as system tests since they will require a maximum of actual (non-mocked) modules. |
| **Performance tests** | Low (excluding dedicated performance campaigns) | Automated | Critical APIs | 20% | Possibly automated in CI in DEV but also manually launched by the developers |
| **Accessibility tests** | Average | Automated + manual | UI | 50% | Axe-Core tests launched in CI to complete with a manual audit |
| **Security tests** | Average | Manual | All | Low, only on sensitive functions | Audit to be scheduled |
| **System tests** | Low | Manuals | UI and batches | 10% | Tests carried out by the development team covering full functional scenarios. The goal is here to test the operation of all the modules (which cannot be automated) and to detect as many bugs as possible before UAT tests. |
| **UAT (acceptance) tests** | High | Manuals | UI, hand-launched batches | from 30% to 80% depending on the number of scenarios planned | Tests carried out in acceptance by some end-users on an uncapped environment with test books. End-to-end acceptance tests (we follow a test booklet with nominal cases), Exploratory tests (we try all possible combinations with minimal guidance in the test booklet) |

For a large project, the test strategy is usually the subject of a separate document. A standard strategy can also be defined at the IS level.

## Ecodesign

List here the software measures to meet the ecodesign requirements previously listed. The answers to its problems are often the same as those to the performance requirements (response time in particular). In this case, just refer to it. However, ecodesign analyzes and solutions can be specific to this theme. Some proposals that can help to save energy:

* Use profilers or development tools integrated in browsers (like Google Dev Tools) to analyze the consumption of resources (number, duration and size of requests).
* For apps, use battery consumption monitoring tools like Battery Historian.
* Use the specialized Greenspector analysis suite.
* Measure the power consumption of systems with PowerAPI2 probes (developed by INRIA and Lille 1 University).
* Measure the size of images and reduce them (lossless) with tools like pngcrush, OptiPNG, pngrewrite or ImageMagick.
* Optimize memory and CPU consumption of applications, tuner GC for a Java application.
* Use lazy loading for occasional resources.
* Limit the results returned from the database (pagination).

An interesting starting point for managing carbon impact can be the Software Carbon Intensity (SCI) formula:

SCI = ((E \* I) + M) / R

Where:

* E (kWh): The total energy consumed by the software;
* I (gCO2/kWh): The amount of carbon emitted per kWh;
* M (gCO2): The carbon footprint of the hardware;
* R: The reference quantity (e.g., per user, per device, …​).

Example 1: The gulp application building process will apply an image size reduction via the imagemin-pngcrush plugin.

Example 2: Robustness tests running over several days will be performed on the mobile application after each optimization to assess the energy consumption of the application.

Example 3: The performance campaigns will integrate a detailed analysis of the consumption of bandwidth and CPU cycles even if the response time requirements are covered to help identifying eco-design optimizations.

Example 4: For a requirement of a maximum of 100kg CO2/year emitted by online service A, which uses 20% of a physical server, given that the carbon intensity of the server, excluding operation, is estimated at 1.5 TCO2 over its entire life cycle of 10 years (thus 30kg CO2/year pro-rata for service A), how much energy (E) can be consumed?

Using the SCI formula (see above), and assuming a total consumption of the server of 800W and 20,000 calls per hour on average, with US electricity having a carbon intensity of 400g/kWh, E must not exceed 175 kWh/year, which equates to 1 mWh/call.

## Management of robustness

### Transaction management

List here the decisions taken regarding the management of transactions. This is especially useful for a distributed system. Some examples of issues:

* Are updates allowed on multiple components during the same request?
  + If so, do we ensure the ACID character at all (via XA mode for example)?
* What transactional engine do we use?
* What level of transactional isolation (read commited, uncommited, repeatable read, serializable)?
* If no transactional monitor is used (call of several REST services in update for example), are there any compensatory transactions in the event of failure of one of the updates?

Example: Our resources are not transactional (REST services), and wanting to avoid making compensatory transactions, it is forbidden to call two update services synchronously. If absolutely necessary, we will use an Event-Driven architecture using a queue.

### Session management

How are HTTP sessions managed to provide an execution context to a user (example: a shopping cart)?

Note that this is primarily a problem for classic web applications whose presentation is generated on the server, not for Single Page Application (SPA) applications which manage all presentation and state locally in the browser.

The choices made here will affect the [infrastructure decisions](view-infrastructure.adoc). For example, if a session is required and the infrastructure is clustered, it will either be necessary to set up session affinity on the servers to force each user to always reach the same server, or to set up a distributed cache allowing servers to share sessions for all users (more complex).

Examples of points to be addressed:

* What data should be kept in session? (pay attention to the volume, especially if the cache is distributed)
* Should the code be thread-safe (if the same user opens another tab in his browser for example)?

Example: our JSF application will store in an HTTP session only its shopping cart, not the product references.

### Error handling

How do we deal with errors? Examples of points to be addressed:

* Do we differentiate functional errors (expected functional errors) and technical? Provide a class diagram.
* How do we log errors? what level of log?
* Where are the exceptions caught? Near the faulty code or in a centralized point of code (like an Error handler)?
* Are we using the language’s standard exceptions (IOException, …​) or our own set of exceptions?
* Is the list of errors consolidated? documented?
* Are error codes assigned?
* Do we display full stack-traces? if so, server side and client side?
* Do we manage retries? if so, how long do we wait between retries (exponential backoff, jitter)?
* How do we manage timeouts?
* How do we manage functional discards ? (i.e. what to do with partial or erroneous requests?)

Example (Spring): technical (unforeseen) errors such as the timeout to a REST service call are caught at the highest level of the application (via an ErrorHandler). All of its information is logged with the full stack-trace but the caller must only retrieve the generic error code XYZ without the stack-trace (for security reasons).

### Frontend robustness

Like the backend, the frontend requires significant robustness, especially since it is in direct contact with the Chair-To-Keyboard interface.

Among others:

* Think about prohibiting double submissions (double call to the backend if you double-click on a button). This does not exclude carrying out hardening checks on the backend side.
* In order to avoid subtle problems (especially when using browser storage such as local/session storage), remember to prevent the same web application from opening in several browser windows or tabs. If attempted, display an error message in the supernumerary windows.
* Always check browser compatibility, even in a controlled environment. If an attempt is made to open a page by an unsupported browser, display an explicit error message on the screen.

Example 1: If the application is opened with IE, an error message should prompt the user to use a supported browser.

Example 2: All buttons in the application must prevent double submission by temporarily disabling buttons when an event occurs.

## Configuration management

How do you configure the application? Examples of points to be addressed:

* What are the variables included in the final package statically?
* What parameters can be changed at runtime?
* Can my application be configured via feature flags (for canary testing reasons for example)? if so, how to handle them?
* In what form are the parameters injected into the application (environment variable? .properties file, database, …​)?
* Does the application accept a live modification?
* Describe the configuration system.

Example (application deployed in Kubernetes):

The configuration will be injected at runtime via environment variables provided in the Kubernetes Deployment Descriptor. No live reloading.

## Branch management policy

What are the branch workflows to plan? git-flow? TBD (Trunked-based Development)? other?

Example:

* The general policy adopted is the [TBD](https://trunkbaseddevelopment.com/) (Trunk-Based Development)
* The main branch is develop. This is a protected branch to which commits cannot be pushed. Any commit will have to be the object of a Merge Request (MR) before integration into develop. The quality criteria (automatically evaluated during continuous integration) must be met for the commit to be integrated.
* Each feature, significant refactoring or bugfix will therefore be carried out on a dedicated topic branch.
* A maintenance branch will be pulled on each x.y version tag. Only bugfixes will be merged into maintenance branches from develop via cherry-pick.

## Versioning

What is versioned and what versioning scheme is uses?

Example:

* In general, any non-derived resource (source, tool, ci-cd script, template, database DDL, …​) must be versioned.
* The modules will be versioned according to the numbering x.y.z (<major). <Evolution>. <fix>)
* The libraries will be versioned according to the same numbering as the modules but the x value will be incremented during any version upgrade breaking upward compatibility (principle of Semantic Versioning).
* The overall logical version of the project will be: <lot>.<Sprint number>.<Deployment number>

## Concurrency

How do we manage concurrent access? Examples of points to be addressed:

* What scope for the objects (if using an IoC engine)?
* Should objects be thread-safe?
* Which methods should be synchronized?
* Risks of race condition? of starvation? dead locks?

Example (Spring MVC): All controllers will be in singleton scope and therefore must in no case store state in their attributes to avoid race conditions.

## Encoding

What are the rules for encoding strings? This is a recurring problem. This problem is, however, relatively simple to solve and requires only rigor. See the examples below for examples of actual measures.

Example 1: The only encoding allowed in all modules and technical components is UTF-8. The use of ISO-8859-1, CP-1252 or any other encoding is strictly prohibited. This includes the configuration of application servers (Node, Tomcat …​), sources, configuration files, databases, and files.

Example 2: If an external system requires sending or receiving character strings in an encoding different from UTF-8 (example: a REST service which returns data in ISO-8859-1) and that it is not possible to modify the contract, it is imperative to translate character strings within an anti-corruption layer as early as possible. In addition, never persist in our systems a data in a non-UTF-8 encoding.

## Timezones

How do we manage the storage of dates? This, as the management of encoding is a recurring problem (one day shift, bugs during summer/winter time changes, etc.) and yet simple to solve: follow the [ISO 8601 standard](https://en.wikipedia.org/wiki/ISO_8601) ("Time zones in ISO 8601 are represented as local time (with the location unspecified), as UTC, or as an offset from UTC."[Wikipedia]).

Example 1: Hours will never be stored without a time zone. Basically, we will use timestamps with timezone (timestamptz) and in Java or JS, objects integrating the time zone explicitly (eg: Instant and not LocalDateTime in java) or epochs. The precision will be at least a millisecond.

Example 2: Dates and date-times will be stored in database as epoch millis in long integer format. In the case of dates, we will store the epoch millis at 12:00 UTC (and not 00:00, too close to the previous day, risk of bug).

## Log management

The log infrastructure aspects are detailed in the [infrastructure view](view-infrastructure.adoc).

Give here the general rules concerning the application traces (logs), the levels and quantity of logs. Think about the use of logs, especially on the server side. Ask yourself if it will be possible to benefit from it in the event of an error in production in the middle of MiB or even GiB of other logs and n threads logging in parallel.

### General rules

Example 1:

* Do not leave development logs in the code (example: console.out("entry in method x") or e.printStackTrace())
* Remember to use discriminating character strings (example: error codes or tags like [APP001] ) to facilitate filtering in the log search tool.
* Always provide entity identifiers and a maximum of context allowing to find the concerned objects.
* Use correlation identifier between third parties (example: processing id generated on the client side in JS, passed to the server).
* Never split a logs into several lines.
* Allow live reloading of verbosity level (useful in production to enable temporary DEBUG logs).
* Avoid expensive calculations (example: many concatenations) and use conditional blocks (example in Java:

if (isDebugEnabled()) {  
 logger.debug (a+b+c)  
}

### Levels and quantity of logs

Explain when and what to log in so as to produce logs that can be used in production.

Example:

Log levels

| Severity level | Context of use | Indicative volume | Environment |
| --- | --- | --- | --- |
| DEBUG | In a development environment, it allows you to display the values ​​of variables, method I/O, etc. | Max a few MiB/ minute | DEV, Testing. Prohibited in PROD unless expressly requested by the project |
| INFO | Start/end of a batch or a call, loading of a new property. Can be used in condensed form for service calls (logging of a call and its context). This is the level of verbosity used for metrology. | Max 10 logs/sec, a few KiB/minute | All |
| WARN | All warning messages about unexpected functional information | No limits but do not abuse them and position as much contextual detail as possible | All |
| ERROR | All errors that do not prevent the application from working. | No limits. Add a maximum of context detail | All |
| FATAL | All blocking errors for the application (BDD access problem, HTTP 404 or 500). Position a maximum of context detail. Remember to log these errors on an appender console in the event that writing to FS is impossible (disk full). Remember that during a fatal error, even writing the log can fail (for example in the event of a memory overflow). | No limits. | All |

## Administrative tools

Should the app provide administration services? It is strongly recommended (this is the factor 12 of the [Heroku Twelve factors](https://12factor.net/)) to integrate the administration code directly within the business code.

Examples of points to be addressed:

* Do I have to provide a way to purge data, logs, caches, …​? (this type of services is sometimes called an 'internal tasks')
* Do I have to provide application indicators for monitoring? (number of files consulted, …​)?
* Do I have to provide migration tools?

Example: The /internal/maj\_2 service will perform a version upgrade of the data model to V2

## Sorting and Pagination

It is necessary to keep a good fluidity of batch data recovery. The pagination allows to limit the chatter between the clients (GUI and batches) and the APIs. Describe here the paging measures implemented on the client side and on the server side.

Example 1 (Server side)

* API output requests are systematically sorted in ascending order (the default) or descending order. In addition, it will be possible to choose the field on which the sorting is done via another query param.
* In order to limit the number of requests to the API, it returns a limited number of elements (this number can be configured according to the size of the individual elements). This is the query param range containing the number of the page to retrieve + the number of elements of the page. Each API will offer a default value (around a hundred).

Example 2 (Client side)

* The sort must be applied to all the elements in the database, not only to the elements of the last query returned by the server.
* The returned elements will be displayed in block tables (configurable size of an indicative size of around 20 elements).

## Provisioning and DDL updates

Describe how the DDL (Database tables definition) and the initial data (such as nomenclatures) will be managed and then updated.

Example: we will use Liquibase embedded in the war to create and update the DDL of the database. There will therefore be no SQL scripts to launch, the necessary queries will be carried out directly by the application when it starts.

## i18N Management

Describe how you meet the internationalization requirements mentioned above.

Example: The management of dates, plurals, and numbers will be done via the [ICU standard](https://icu.unicode.org/) and the JS library [messageformat](http://messageformat.github.io/messageformat/).