Development Guidelines

Release 0.1.0

Osvaldo

Feb 14, 2022

Table of Contents

This set of pages holds general guidelines and orientation for the development of software projects. All projects must follow all the rules from these guidelines.

CHAPTER 1

General Guidelines

1.1 Simplicity

We appreciate simplicity. Simple does not mean simplistic. It's hard to achieve simplicity because it demands a lot of thinking about the problems and solutions.

1.2 Pragmatism

We are pragmatic about our choices. We will never adopt a solution without a strong rationale about the decision.

1.3 Quality

Quality vs fast-delivery is not a dilemma. We use agile practices to handle this kind of situations.

We cut solution scope, make partial delivery with high quality or try to design an alternative solution that enable an easy refactoring in the future.

In fact we cannot neglect quality but we try hard to deliver on time because we are a business oriented team.

1.4 Craftsmanship

We believe strongly in the Software Craftsmanship movement and in their manifesto.

> As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:
>
> - Not only working software, but also well-crafted software
> - Not only responding to change, but also steadily adding value

- Not only individuals and interactions, but also a community of professionals
- Not only customer collaboration, but also productive partnerships
- That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

—Software Craftsmanship Manifesto

CHAPTER 2

Processes & Practices

Guidelines for development processes and practices.

2.1 Business Orientation

A developer must go beyond software development. It's required that developers understand our business and its problems to create better solutions for it.

2.2 Guideline Enhancement Proposal

None of the guidelines in this document are immutable, but whenever a change or improvement is required, just fill in an Enhancement Proposal at our internal Wiki and follow the process described at GEP-0000 until approval (or rejection).

You can use GEPs to propose technical solutions to common problems.

2.3 Code Review

Code Review is a must-have discipline that improves the quality of our software and, mainly, spread the knowledge about our platform across the team. No code goes to production without passing the code review process.

To make a good code review we must pay attention to the following aspects:

Be propositional in the reviews It means that instead of saying "your code is poorly done, redo this", it's better to say "your code does not cover such and such scenarios, you probably will need to implement these new behaviors". Assume you got it wrong: Always remember that you are reviewing the code, not the human behind it. Always ensure you are addressing the code, not the author. If something looks strange or wrong, always assume **you** did not understand it and ask for clarification.

Defend your arguments, but do not forget you can be wrong. Whenever giant egos meet each other, it makes people disperse from main goal of interaction. In our case, the goal is to solve some business problem or opportunity with software. So to avoid unnecessary conflicts and keep the whole team focused on what matters, be careful when choosing your arguments' wording and the way you criticize others in code reviews. Practical example: instead of saying "this feature or code is irrelevant", it would be better to say "I do not understand why we're prioritizing this feature right now, could you explain?". It's OK to agree or disagree. We all have our own preferences and tastes. Look for a middle ground and remember that you own the code as much as the other person on your team.

Communicate with other people, not with your computer screen. This is completely basic, but especially working remote, sometimes we do not realize that we are talking and interacting with other people. Greeting and being kind can be a great way to get the reviewers' attention to your code. It's always good to remember that written communication is hard. Even with extra care, someone could misunderstand something. So, instead of assuming something, just ask twice before you choose not to be polite. Be human.

Show mistakes, make suggestions but also praise good code. If you see some improvement to the code you are reviewing you should make suggestions for improvement. If you see something wrong with code, request a fix. If you learn something with code or if the code shows quality and good practices, praise it.

2.3.1 Pull Requests

We use Github Pull Requests as a tool for code review.

Make pull requests as small as possible while still delivering value. Pull requests with a lot of changes usually are left behind by reviewers. Whenever the change involves too many files or lines, split it in smaller changes and open more than one pull request. You can still open several pull requests that are dependent on each other and work to approve one at time sequentially;

Provide relevant and important information for your reviewers. A reviewer needs to find all relevant and important information to review the code in the Pull Request description. Although we can link issues from bug/task systems, it's much mentally easier and inviting for the reviewer to find all information directly in the PR description. At the same time, do not over describe the PR nor be too wordy. Remember that the reviewer still needs to read all the code that you submit (no one told you that it would be easy);

Try to deliver the best code possible from the very first Pull Request. Sometimes the idea of submitting a PR with a code that isn't very good can be tempting, since it will be reviewed and you will have other chances to improve it. Poorly implemented or too complex code will only result in negative reviews and more iterations (review, request, change, submit). It will demand more time of the whole team, it will make the process more tiring and probably the final code produced will not be so good;

Use the GitHub compare feature to get feedback instead of opening "WIP" Pull Requests. It's very usual to request some pair feedback about the code you're making. However, you do not need to open a new Pull Request labeled "work in progress" to do it. Use the GitHub compare feature, in which any branch can be compared to the master branch. For example: if you pushed your code to a new remote branch called `my-feature` in `foobar-api` repository, the compare link would be `https://github.com/[org]/foobar-api/compare/my-feature`.

Work to have your Pull Request merged. The developer who opened a Pull Request (the author) cannot merge it, but it is his responsibility to get it merged, so the author must argue in favor of their Pull Request with the other developers to convince them to merge it. All Pull Requests require at least two approvals to get merged, but at least one of these approvals must come from a member of another team or squad. If a reviewer wants to block a merge, the "Request for Changes" GitHub feature must be used, otherwise Pull Requests with two approvals can always be merged.

Pull Requests labels

do-not-merge If your Pull Request has other PR as a dependency or the changes need to be deployed in a specific time, mark it with the `do-not-merge` label. Pull requests in `do-not-merge` still need to be reviewed, but CANNOT be merged. You can use this label in Pull Requests with migration code which deployment cannot be done immediately;

wip If one of your opened Pull Requests has a change request that will take time to implement, mark it as a wip so reviewers will understand that still has work being done. Important: use it wisely, because it's an exception, not a practice.

2.3.2 Commit Messages

Commit Messages must be written in english using the imperative mode in the summary line:

```
Fix order cancellation bug #123
Add new publication status (sent)
Change Order.is_active() behaviour in case of blocked status
```

You can read more about good commit messages in the following articles:

- https://github.com/erlang/otp/wiki/Writing-good-commit-messages
- http://chris.beams.io/posts/git-commit/

2.4 Pair Programming

We encourage pair programming as a practice that improves solution design, speeds up the integration of new developers into the team, and allows more experienced programmers to help those with less experience.

Although we encourage Pair Programming, we don't require it and won't force anyone to do it.

2.5 Continuous Integration

All code submitted to a Code Review and merged at master branch of a repository must pass all checks and tests under our Continuous Integration environment.

Our continuous integration must run the following checks:

1. Run all automated tests;
2. Check *Coding Style*;
3. Run linters to check the presence of credentials, debugging artifacts, etc.

2.6 Deployment

Todo

procedures for deployment, deployment follow-up, production readyness (monitor, backup, credentials, etc), checks, etc

2.6.1 Continuous Deployment

Todo

procedures for deployment, deployment follow-up, production readyness (monitor, backup, credentials, etc), checks, etc

2.7 Scheduled Maintenance

Todo

TODO

procedures for scheduled maintenance...

2.8 Service Unavailability and Disaster Recovery

Todo

TODO

procedures (maintenance mode on, communicate stakeholders, turn queue consumers off, recover data from objects history when it exists, recovery remaining data from backups, put services back, maintenance mode off, communicate)

2.9 References

- Anatomy of a Code Review
- Yelp Code Review Guidelines

CHAPTER 3

APIs (WIP)

Application Public Interfaces (API) are channels through which multiple software components communicate. A good API provides efficient communication between components and is easy to be used by the developers that create these components.

This chapter provides guidelines for creating APIs with these characteristics for services. If you need more informations about APIs in the context of libraries take a look at chapter *Libraries and APIs*.

This chapter is based mostly on guidelines created by PayPal, Google, and Microsoft. If you have some question about one subject not covered here, we recommend these documents as a further reference.

Todo

This chapter contains only the Table of Contents with the topics that will be covered.

3.1 Design

- Design First (required)
- Aspects:
  - Ease of use
  - Single Responsability (loose coupling, encapsulation, cohesion etc)
  - Robustness (consistent, stable, contract-based etc)
  - Security
- Design Methodology
  - Top-Down (from client to API) (recommended)
    - Use the client use cases to guide your API design. When you create a client interface you will see what informations must be shown. Therefore, you can provide an API that returns this information in one request.

  - Bottom-Up (from data models to API)
- Architectural Styles for Service APIs
  - REST (recommended)
  - CQRS
  - RPC (XML-RPC, SOAP etc)
  - GraphQL
- Hypermedia (optional)
- Naming Conventions and Standards
- Resource
  - Resource is more than a data model
  - A list of resources is one resource
- Resource Representations (serializers, Content-Types etc)
- Resource Location (URL)
  - Schema
  - Flat is better than nested
- Versioning
  - API Lifecycle
  - Deprecation policy
  - Future proof APIs
    * Designing extensible APIs
      · When you decide to create a boolean "flag" on your resource, stop and think again. Is it possible to change this binary parameter on a variable parameter? Eg. free_shipping=True. Free shipping does not exist. There is someone paying for this shipping. Why not model configuration like:

      ```
      {
        ...
        "shipping_payment": [
          {
            "payer_type": "seller",
            "rate": 0.5
          }, {
            "payer_type": "carrier",
            "rate": 0.2
          }
        ]
        ...
      }
      ```

      to tell application that seller will pay 50%, carrier will pay 20% and buyer will pay 30% for the shipping.
    * Backward compatible modifications
    * Backward incompatible modifications
  - Multiversion selection and management

- Company-specific API Philosophy
  - HTTP shines! Use it.
  - Postel's Law of Robustness
  - Reactive APIs
    - Create APIs that "reacts" to events. Eg. If we set the field "approved_at" with a timestamp in one order it's clear that it must be transitioned to "approved" status. Same for "invoiced_number" -> "invoiced". Do not allow forced transitions like PATCH /order/id {"status": "invoiced"} or pass multiple arguments to do this transition like PATCH /order/id {"status": "invoiced", "invoiced_at": ..., "invoice_number": "..."} (this use case is extremely error prone).
  - Future proof
  - Check, recheck, double check and check again for every "status" and state machines on resources. There are lots of "gotchas" on state names and transitions.

3.2 Specification and Documentation

- Specification Tools
  - Swagger
  - Pactum
- Documentation
  - Types of documentations
    - Usage Manual
    - Tutorials
    - Use Cases
    - Reference
    - Implementation Documents (private)
  - Tools
    - Pactum Documentation toolchain
    - Sphinx

3.3 HTTP, REST and Web

We love the Web and HTTP protocol. The simplicity of the concepts like Resource/Document, Resource references hyperlinking (through URL), and the stateless model of Request/Response forces the result of solutions design to be simple (but not simplistic). We believe that RESTful APIs embraces this simplicity.

- Resource Representations
  - JSON
  - Protobuf
  - HTML (required for "Web APIs")
- Resource Locators

- No trailing slash at URL path: /resources instead of /resources/ (backward incompatible, support HTTP 307/308 redirects on server and clients)
- Resource names on path must use plural for collections and singular for single resources. (backward incompatible)

- Web is an API, Web as an API
- Request
  - Methods
  - HTTP Headers
  - Data model and representation (serialization)
    - Data types (date, timestamp, status enum, nil/null etc)
  - Company "way of REST"
    - Path version selector
    - Filtering (querystrings)
    - Searching (querystrings)
    - Pagination (querystrings)
      - Always set a default and a max limit for limit and page size
      - limit/offset (required)
      - page/pagesize (required for "Web APIs")
      - Hypermedia links to "next" and "previous" pages
    - Fetch control (querystrings)
    - Bulk Requests support with multipart content
    - PUT As Create
    - Asynchronous Request/Response
    - Custom HTTP Headers support
      - `X-HTTP-Method-Override`
      - `X-Request-Id`
    - Idempotent POST, PUT and PATCH (303/304)
    - JSON PATCH support
- Response
  - Status Code
    - Ranges
    - Allowed Status Codes and their Usage
    - Method x Status Code Mapping
  - HTTP Headers
  - Error response data model
  - i18n & l10n

        * Error messages must be returned based on `Accept-Language` request header for error messages or resource data translation (eg. Product name translation). It's recommended to return the original message template string and error data inside separated object to allow client developers to create custom translations:

```
# No Accept-Language or unknown language
400 Bad Request
{
    "length": [
        {
            "message": "Invalid minimum length 6.3in",
            "error": {
                "message_template": "Invalid minimum length {size}{unit}",
                "data": {
                    "size": "6.3",
                    "unit": "in"
                }
            }
        }
    ]
}

# Accept-Language: pt-br
400 Bad Request
{
    "length": [
        {
            "message": "Comprimento mínimo inválido 16cm",
            "error": {
                "message_template": "Invalid minimum length {size}{unit}",
                "data": {
                    "size": "6.3",
                    "unit": "in"
                }
            }
        }
    ]
}


# Accept-Language: [weighted list of languages]
... most weighted language available ...
```

    - Hypermedia
        * Link description and relations
        * Links Array
    - Company-specific standards
        * Asynchronous Request/Response
            · Sync vs Async with state control to keep response time low
        * Not Found instead of Forbidden for anonymous access

3.4 Implementation

- Response time
  - Fast is better than slow
  - Execute performance and load testing in all endpoints of API before every deployment
  - Default maximum response time constantly checked on monitoring
  - Avoid caches. Again, avoid caches. If it's required your app must also work without it (slow response time instead of errors)
- Security (SSL, auth&auth etc)
- Protection (throttling, DDoS protection etc)
- Implementation details protection (hide database sequential pk from URLs, don't return database errors on error messages, never run debug mode on production environment etc)
- Event triggering
- Deployment checklist

3.4.1 Denormalization and Data Sync

Todo

write this topic...

3.5 Common Solutions

Standard techniques to solve common problems:

- Use PUT-as-create with an client-side generated ID to fix duplicated resource
- creation caused by double-clicked issues on client web application.
- Use status fields to manage workflows of objects that need to be processed on multiple steps.

3.6 References

3.6.1 API Design Guidelines

- PayPal API Standards
- Google Platform API Design Guide
- Microsoft API Guidelines
- Zalando RESTful API Guidelines
- PayPal security guidelines and best practices
- Interagent / Heroku API Guidelines

3.6.2 Articles

- The definitive guide for building REST APIs

CHAPTER 4

Architecture

Important: Ready for revision.

Good architecture is a important subject. This section will describe some guidelines that must be followed when architecting a product or solution.

4.1 General Rules

4.1.1 Minimal Dependencies

- Reduce the number of requirements and components for a project. Less "moving parts", less complexity. Less complexity, less bugs.
- Resist the temptation to add another element to the solution stack.
- Limit the use of new tools to occasions where current tools are insufficient.
- Adopt new tools only when they are beneficial to the project.
- Prefer third party (managed) tools in cases where the solution is not part of our core business. Example: if you need a solution for message queueing prefer using AWS SQS instead of make a RabbitMQ deployment.

4.1.2 Auditability

- All transactions need to be traceable. We need to know *When* (timestamp), *Who* (user), and *Where* (source) started *What* (transaction).
- Transactions must be uniquely identified (`transaction_id`) through all components of platform.
- Transaction identifier must be present in all logs (see *Logging*).

- It's important to make a distinction between Transaction ID that relates with a business transaction (eg. product approved by moderation) and Request ID that relates with a implementation detail (HTTP request).

4.2 Microservices (or SOA) Architecture

We deploy products and solutions as a bunch of highly specialized and reliable services that communicate each other using messages.

After some time deploying this kind of service we have detected some building blocks and patterns for architecture.

4.2.1 Building Blocks

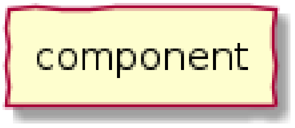
Message

Messages are the base building block of our architecture. Every service communicate with each other using messages.

Messages follows a common contract and must be serialized using a open-standard serializer like JSON or Protobuf. You can wrap this messages with some metadata.

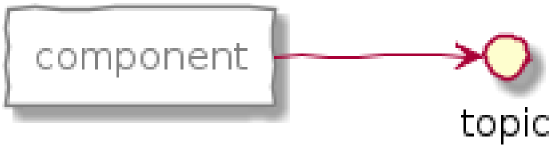
Component

Component is a service or API that receive, processes and triggers events. It's implemented and deployed as software processes.



Topic

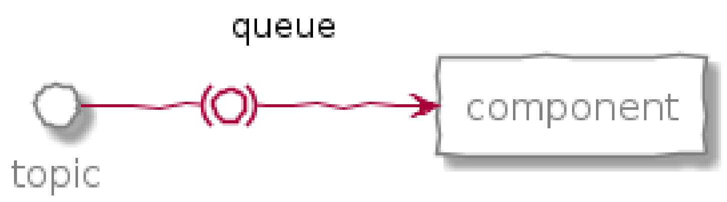
Our architecture use topic as a location where components send messages (Publishers) that would be listened by other components that subscribes to it (Subscribers).



Topics belongs to the platform, ie, any component can post messages because they are public (to the platform) and global.

Queue

Every component that needs to listen for messages published on topic (see *Topic*) must use a queue as a topic subscriber.

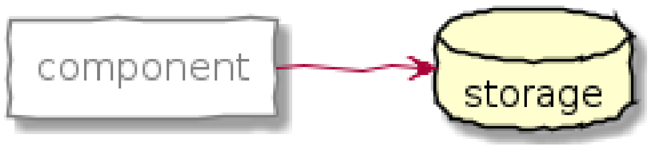


Queues belongs to the component (eg. *Service* or *Broker*) that subscribes a topic. Unlike topics, queues are private and local to the component that consume its messages.

It is very common that different components listen to the same topic. Assigning one queue to each component and knowing that each queue receives a copy of the published message we can guarantee that one component won't process other components messages.

Storage

Storage is the location where we store validated and consistent data.



We usually use relational databases (see *Database*) to store data at our platform.

We PostgreSQL, a lot (you should not use anything different).

4.2.2 Patterns

We can connect the building blocks above to create patterns with specific responsabilities in our architecture.

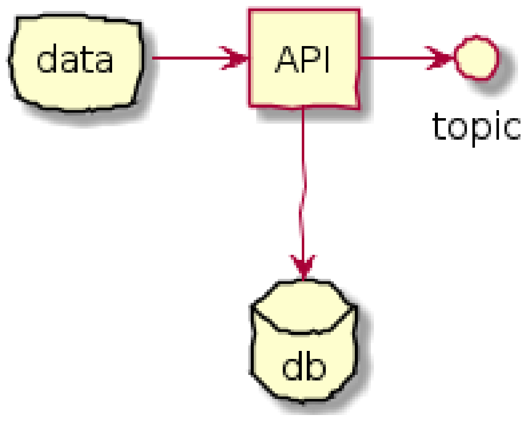
API

The APIs are the channels which data is inserted and retrieved from our platform.

The responsabilities of an API are:

Data input and recovery

Our APIs are made available mostly using the REST model with JSON serialization using the HTTP protocol.



Data validation (including state transitions)

All data sent to our APIs must be valid and APIs need to be able to validate data autonomously, ie, APIs cannot request informations to other APIs (see *Denormalization and Data Sync*) to validate data.

Some resources of our APIs provides fields that stores status/state info. It is responsibility of API validate these status and their transitions.

Data persistence

The persistence/storage of data is also a responsibility of the APIs.

As we already mentioned, we use a relational database in all cases where it is not absolutely necessary to use another type of storage.

This persistence must be wrapped by a transaction with (see *Event triggering*) and rolled back in case of failures. API must return an error in these cases. Like in the following pseudocode:

```
transaction = begin_transaction()
try:
  persist(object)
  trigger_event(object)
except:
  transaction.rollback()
transaction.commit()
```

Event triggering

Once the data is persisted APIs need to trigger an event reporting this fact by posting a message on a specific topic (see *Topic*).

The payload of the event must include the content of the persisted object or, at least, a reference to the object at an API.

You can use the following payload as an example for the content of the event message:

```
{
  "transaction_id": "deadbeef",
  "object_type": "order",
  "object_id": "bb654446-22d4-4f28-ab3e-e72bebb89a8c",
  "href_template": "https://api.example.com/{object_type}/{object_id}"
  "href": "https://api.example.com/order/bb654446-22d4-4f28-ab3e-e72bebb89a8c",
  "action": {
    "type": "update",
    "changes": [
      {
        "field": "status",
        "value": "invoiced",
        "old_value": "new"
      }
    ]
  },
  "embedded": {
    "order_id": "bb654446-22d4-4f28-ab3e-e72bebb89a8c",
    "seller_id": "9d054c45-a72e-4878-a932-f131e92e2bf7",
    "status": "invoiced"
  }
}
```

- `transaction_id`: used to make transaction traceable (see *Auditability*);
- `object_type`: the type of the object that received the action that triggered the event;
- `object_id`: the ID of the object that received the action that triggered the event;
- `href_template`: the template that you can use to generate the hyperlink reference to the object. You can use it to generate custom URLs to access an specific objects;
- `href`: the hyperlink reference to the object (for convenience);
- `action`: the action that triggered the event. In the example we can see a change (`update`) in the order. Based on the list of changes we can also see that the order's status transitioned from `new` to `invoiced`;
- `embedded`: some fields of the object that could be directly used by other services. These fields could be used to reduce the amount of requests to the APIs but can also increase the payload of the messages. Use it wisely.

Idempotency Handling

In cases where one of our services make a duplicated request to our APIs it must handle this correctly. A duplicated *POST* request must receive a *303 See other* response and other request methods must receive a *304 Not Modified* response.

The implementation of this handling depends on specific business rules. But let's look for some examples.

Sending the same *POST* that creates a transaction twice:

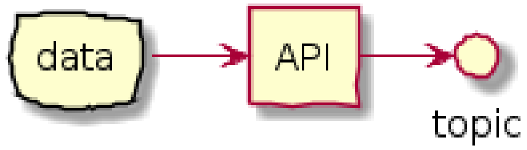
```
$ curl -i -X POST https://api.example.com/transaction/ \\
       -d '{"transaction_id": "deadbeef"}'
HTTP/1.1 201 Created

$ curl -i -X POST https://api.example.com/transaction/ \\
       -d '{"transaction_id": "deadbeef"}'
HTTP/1.1 303 See other
Location: https://api.example.com/transaction/deadbeef
```

Change an order status that is already in *invoiced* status:

```
$ curl -i -X PATCH https://api.example.com/order/XYZ/ \\
       -d '{"status": "invoiced"}'
HTTP/1.1 304 Not modified
```

Webhook Handler

A webhook handler resembles an API except that it does not persist data and is not required to adhere to the *APIs (WIP)* guidelines.



Webhook handlers exists to receive notifications from external partners. It is important that all webhook handlers work together with a scheduled job service that retrieves notification data that was lost due to failure on notification handling.

Service

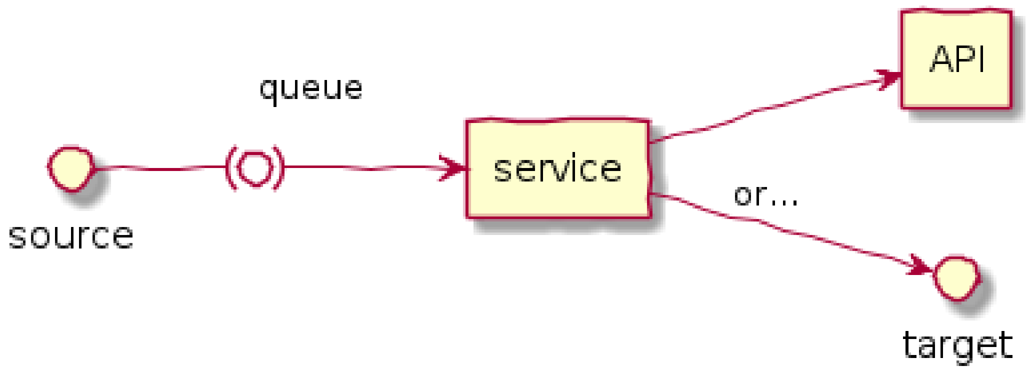
Services (also called as Workers or Consumers) are components that process (consume) messages. These messages are sent to queues that subscribe to topics. You can also read this as "the services listen and process messages from topics".

One service consumes messages from one queue, as an input data, processes these data and then generates an output as a publication on topic or an API request.

The simplest type of service are the 'de-queuers' that basically process messages from a single queue (that subscribe a single topic).

So a service works following the steps below:

1. Get *one* message from a queue (that subscribes a topic);
2. Process this message (following/applying business rules);
3. Get extra informations requesting them to APIs (optional);
4. Send the result publishing it in a topic or posting *one* request to an API.



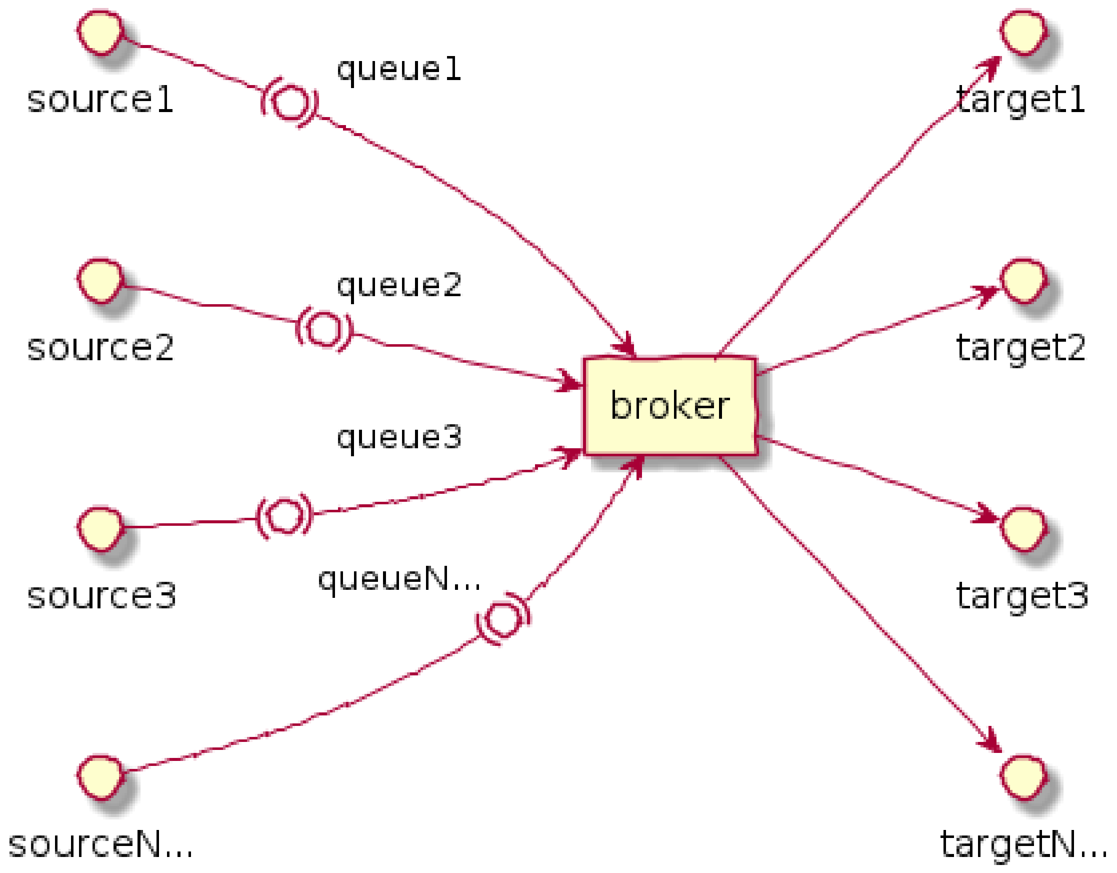
The only reponisibility of a service is: **Business Logic**.

We implement most of the business logic of our platform in services. This design allows us to keep API agnostic about specific business rules.

This approach allow our APIs to be used by other market players, and also allow us to build services with different business rules for other markets.

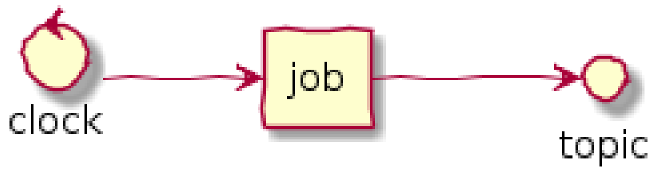
Broker

Broker is a special kind of service that consumes more than one queue. We use brokers basically to make code maintenance easier grouping several services that interacts with, eg, one API in a single code base/deploy.



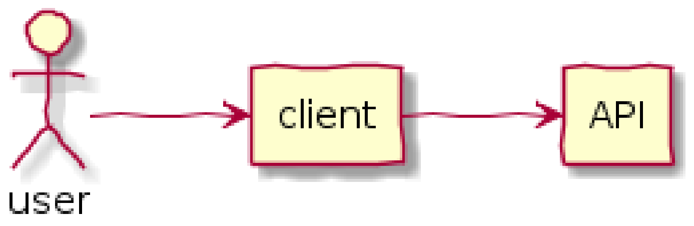
Scheduled Job

Scheduled Jobs are services triggered by the clock (usually in a regular cycle) to make some kind of batch action and publish the results in one topic (eg. get all orders lost by webhook handler and publish one-by-one in a topic).



Client Application

Client Applications are web (or mobile) applications which provides the means by which users interacts with our platform.



4.3 Integrations

We've two kinds of integrations at our platform:

1. **Internal integrations:** when one of our components must interact with other component of our platform (eg. service makes a request to an API) and;
2. **External integrations:** when one of our components must interact with a component of other platform (eg. service makes a request to one of our partner's API).

On both integration scenarios we need to start from the following premisse:

> No matter if a system is internal or external it eventually...
>
> - ... goes **offline**...
> - ... **crashes**...
> - ... or **change their behaviour without notice**.

So, to make an integration work in a reliable fashion we need to follow some rules and procedures:

- Be prepared for the worst;
- Create a SLA for all integrations;
- Monitor (see *Monitoring & Logging*) all aspects of integration (eg. errors, performance, availability, etc);
- Always use a Circuit Breaker pattern for integration;
- Set a (small) timeout for requests to avoid that the client becomes blocked;
- Create a retry policy based on defined SLAs or based on informations at error response (eg. *Retry-After:* HTTP header in *503 Service Unavailable* responses);
- Remember that, depending on the context, some errors are recoverable and others are not recoverable. Handle error responses appropriately: retrying, rolling back, logging, etc;
- All these rules and procedures must be implemented out-of-box in all services. No code deployment must be required to handle unavailability scenarios.

4.4 References

- Some Guidelines For Deciding Whether To Use A Rules Engine

CHAPTER 5

Monitoring & Logging

This chapter you will found informations about monitoring and logging.

5.1 Monitoring

Todo

write it...

5.2 Logging

This guildelines were built upon the concepts of 12 Factors App and Splunk Logging Best Practices.

We generate logging based on the transactions processed by the system. The definition of *transaction processing* according to Wikipedia:

> Transaction processing is designed to maintain a system's Integrity (typically a database or some modern filesystems) in a known, consistent state, by ensuring that interdependent operations on the system are either all completed successfully or all canceled successfully.

Examples of transactions in diferent contexts:

- Database: database transaction (commit/rollback);
- Web Application/API: request/response cycle;
- Worker: process a message;
- Business Transactions: bill a credit card, cancel a contract.

General practices for logging generation:

- You should use the framework/language tools for logging (eg. python's `logging` module);

- You should not use `print()` or `echo()` to produce log messages;
- You must start your log entry with a timestamp using UTC timezone.
- You should use a standard output device to produce log (`stdout` or `stderr`);
- Errors in code (programming errors) should be handled apart of business failures (transactional failures). See *Error Reporting*;
- You should not produce multi-line log messages (`\n`) for non-debugging logs (see *Log Format*).
- It's recommended to take care about the amount and the relevance of log you generate to avoid the blindness caused by excess of noise in logs and to reduce the costs of storing it.
- No sensitive or private information could be logged. You should mask all all informations that need to be protected by the Terms of Service of our product or GDPR laws. Informations like secret keys, passwords, financial informations, personal informations and implementation details of our system or infrastructure (eg. database/database table names, full paths of deployments, etc);
- We recommend that you create an unique ID for each transaction and print it on logs to make it easy to track all sub-transactions and operations inside of a transaction;
- You should use string representations or safely encoded strings in logs to avoid encoding & decoding issues with non-ascii caracters;
- Production environments must enable at least `INFO` log-level. For staging, and local development environments we use `DEBUG` level.

5.2.1 Logging Errors

There are two different kinds of error logs that need to be managed separatadely. The software errors (see *Error Reporting*) must be reported in a specific system for error tracking and, transactional errors, that occurs when something goes wrong with the business rules, must be logged as a regular log with log level `ERROR` (see *Log Levels*).

5.2.2 Log Format

The modern systems for log agregation offers a lot of indexing, searching, and analytics tools to be used by developers.

To make this possible this systems recommends that we generate logs in a structured way. That's why we recommend you to use JSON-serialized log messages.

- Send the plain JSON-serialized string in a single line for each log record.
- The log structure must contains at least the following information:
  - `LOGLEVEL`: the level of the log message;
  - `TIMESTAMP`: Timestamp in `asctime` format and UTC timezone;
  - `GUID` (optional) - GUID (eg. `UUIDv4` string) of transaction (if available);
  - `FILE/FUNCTION:LINENO` (optional): file, function and line number where the log was generated. This information must be included **only** in `DEBUG` log level.

5.2.3 Log Levels

`DEBUG` and/or `TRACE` Detailed information about the whole transaction and it sub-transactions. You can print detailed and verbose information about the internal state of transaction like variables, call trace (in cases where of `TRACE` is supported), etc. It is important to take care of customers' private data and sensible informations.

By default this log level is not enabled in live production servers but, besides that, could be enabled for live production debugging purposes.

`INFO` or `NOTICE` Summarized information about a successfully finished transaction. You should put one or more key information that make this transaction trackable inside the system and you should describe what transaction executed (eg. `operation=bill credit card (capture), customer_id=XYZ123`). This log level should be enabled in live production environments. In cases where the system generates a huge amount of data (eg. request/response log) you could agreggate the information in batches or route the logs to an specific system that can handle these logs in a better way.

`WARNING` Something exceptional happened during the transaction processing but the system was able to recover from this exception (eg. `operation=bill credit card (capture), customer_id=XYZ123, result=timeout connection (retrying #1 of 3)`).

`ERROR` The transaction failed in a way where the system could not recover itself (eg. `operation=bill credit card (capture), customer_id=XYZ123, result=failed after all retry attempts.`). Errors caused by the end user must not be logged as a error (eg. Invalid username/password errors).

`CRITICAL` The transaction failed and the system breaks completely due to this failure. This error shoud be logged in but need to raise an exception to the systems that manages error reports (see *Error Reporting*).

5.3 Error Reporting

Errors in code are caused by some part of the code that is wrongly created by the developer. Usually it raises a language exception that are not handled by the code.

You must not send these errors to the transactional logs (see *Logging*).

5.3.1 Exception Handling Service

We use a service to capture, collect, aggregates and monitor this kind of errors. The system we're currently using for this purpose is Sentry.

CHAPTER 6

Implementation

6.1 Code

Todo

good code > good doc, early optimization trap, early abstraction trap, exception/error handling: (un)recoverable, (un)expected errors

Code guidelines and best practices.

6.1.1 General advices

- Always KISS - Keep It Super Simple;
- All source files must be written in English (variables, functions, classes, docstrings and etc). Only strings submitted to customers/users should be in in their native language (i18n/l10n);
- A code well written is self documented;
- Pay attention to the quality of your code using some indicators like cyclomatic complexity or the presence of some Bad Smell.

6.1.2 Coding Style

Coding style is a complex subject with lots of personal preferences and we believe that this preferences must be respected until the limit where it causes readability issues to other developers in our team and to keep some level of consistency on our code base.

Usually we use the coding style proposed by the community of an specific programming language, eg:

- Python - PEP-8 – Style Guide for Python Code but also consider important writing pythonic code as in Beyond PEP-8 with some customizations (see below).

- Go – we use the coding style applied by gofmt in our Go code
- Elixir - we use the coding styled applied by _mix_format in our Elixir code

PEP-8 Customizations

We use most of the rules defined by PEP-8 except the rules that define the maximun line length. Instead of 80 characters we define a soft-limit in 120 characters. You are allowed to use more than 120 characters but use it with moderation.

6.1.3 Tests

Todo

expand this session (or move to a new chapter), describe some test patterns?

- Unit test every function, method and class.
- Integration tests should assert each part called using mocks or return checks.
- Avoid using VCR-like mocking system. VCR-like libraries store credentials used for tests in fixture files and it creates a security breach. The fixtures created by them are strongly dependent of the API state, making it hard to update the test in the future.
- We don't use test coverage as a metric, but as a way to find use cases not tested.
- Features with multiple layers should be tested on all layers (an API endpoint should have tests in the manager level (focused in the data) and API level (focused in correct HTTP usage)

6.1.4 Configuration

Todo

move this session to other chapter?

- Configuration through environment variables: 12-factor configuration.
- Avoid different configurations for each environment.
- Decouple configurations with libraries like prettyconf.
- Configurations should control only the software behaviour. Business logic configurations must be handled like system data; database-stored and configured through an administrative interface.
- Configurations that frequently change are good candidates to leave configuration files.

6.1.5 Security

Todo

create a chapter specific for security?

- Sensible and secret data must not be versioned with the code.

- Always follow and apply security patches.
- Dependencies must be kept up to date.
- Only use known and tested security methods and systems.
- Security measures shouldn't be entangled with infrastructure.
- Handle HTTP errors with static pages to avoid exploits.

6.1.6 Libraries and APIs

Todo

Informations are temporarily in Portuguese but it will be rewritten in English in final version of the document.

- Devem ter changelog.
- Mudar a versão (major) sempre que houver quebra de compatibilidade retroativa.
- Manter a versão anterior dentro de um plano de "deprecation" definido previamente em cada projeto.
- O modelo de versionamento deve ser adotado consistentemente em todas as APIs de um mesmo projeto.
- Documentação
- Todas as bibliotecas devem ser versionadas segundo as diretrizes de versionamento semântico http://semver.org/ ignorando apenas os sufixos como: pre, rc, alpha.
  - Formato major.minor.patch;
  - Todas as alterações devem ser acompanhadas pela atualização da versão.
- Manutenção de Changelog atualizado.
  - Podemos usar como referência as Definições do Projeto GNU.

6.2 E-Mail

Todo

write this

6.3 Database

Todo

write this

non-sequential ids, strings instead of enums

CHAPTER 7

Books

Suggested reading list:

- Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Margin
- The Pragmatic Programmer: From Journeyman to Master, Andrew Hunt, David Thomas
- Building Microservices: Designing Fine-Grained Systems, Sam Newman
- Refactoring: Improving the Design of Existing Code, Martin Fowler

CHAPTER 8

Document Conventions

We use keywords like “MUST”, “MUST NOT”, “REQUIRED”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” with the same definitions as the RFC 2119.

Machine-readable text, such as code, URLs, protocols, etc are represented with monospaced font (eg. HTTP method `POST`). We use `{}` to delimiter template variables in samples and `#` as a comment mark:

```
# This is a comment
https://api.example.com/account/{account-id}
```

CHAPTER 9

General References

A list of documents and sites we use to produce our guidelines.

9.1 Handbooks

- Basecamp Handbook
- Gitlab Handbook
- Valve Handbook

9.2 Development Guidelines

- Plataformatec
- Terraform Recommended Practices

9.3 Architecture Patterns

- Azure Architecture Center

9.4 Culture Books

- Netflix Culture
- Disqus Culture
- Loadsmart Culture