API Producer Guidelines

These guidelines are provided to properly document, design and implement a REST API.

They are organized into 5 **themes** as point of views that will be the focus of distincts peoples.

Operational rules focus on **SLA** and **how to ensure it**.

> **Hint**
>
> Not all these guidelines apply to your perimeter !
>
> see SG Group API Guidelines and SGM API Wholesale Guidelines first.

Information¶

Properly declare and maintain your SLA information within the API Catalog, on the “Usage” panel.

Complete or update SLA information¶

Uptime indicates at which time your API is supposed to be callable.

Provide valid SLA information about your API as specified in Usage

Uptime examples:

- “7/7 24/24” for an API that never stops
- “5/7 24/24 Paris TZ” for an API that stops on weekend
- “5/7 9h-17h Paris TZ” for an API that stops every day

API-403 manual **Complete or update SLA information** Fields : Uptime, Support hours, RTO/RPO and support contacts are filled accurately	Availability

Stability and Reliability¶

APIs must be engineered in such a way that the development, deployment, the adoption of new technologies, and the decommissioning or deprecation of other services do not give rise to instability across the larger API ecosystem.

This requires putting measures into place to protect against the negative consequences that may be introduced by these types of changes. **A reliable API is one that can be trusted by other APIs and by the overall ecosystem.**

Stability goes hand in hand with reliability because each stability requirement carries with it a reliability requirement (and vice versa): for example, stable deployment processes are accompanied by a requirement that each new deployment does not compromise the reliability of the API from the point of view of one of their clients or dependencies.

There are several things that can be done to ensure that an API is stable and reliable:

- A standardized **development cycle** can be implemented to protect against poor development practices.
- The **deployment process** can be designed so that changes to the code are forced to pass through multiple stages before being rolled out to all production servers.
- **Dependency failures** can be protected against.
- **Health checks**, proper routing, and **circuit breaking** can be built into the routing and discovery channels to handle anomalous traffic patterns.
- Finally, APIs and their endpoints can be deprecated and/or decommissioned without causing any failures for other APIs.

Expose resiliency model¶

Describe clearly your resiliency model and how you manage it, in order to build trust with your clients. Such description can be : directly in your API card description, referenced via the "Additional information link", put in the README.md of API's github repository

Examples: - Single VM in single DC? - Multiple VM in active/active mode using a VIP? - Multiple DC in active/passive mode? - Multiple DC in active/active mode using a GSLB (DNS LB)? - potential downtime periods - expected uptime period (including hours / week-ends) - support availability - Maximum downtime period (RTO) - Maximum data loss period (RPO)

API-384 manual **Describe resiliency model**
Resiliency needs and current solution are set/referenced in review comment or from the Availability API card

Expose scalability model¶

Expose the scalability model to build trust with your clients:

- fixed architecture or scalable
- sizing and redundancy versus the resiliency model (especially up time)
- for API and for storage

This is a good way to warn your clients about the way they interact with you (frequency of calls, volume of data...) and the way they should behave (e.g. if they expect increase of their usage).

This information helps you plan SLA.

API-385 manual **Describe scalability model**
Scalability needs and current solution are set/referenced in review comment or from the Availability API card

Use standard Load Balancer for resiliency¶

In order for your API to be resilient to datacenter loss you should be able to deploy it on several sites and have a load balancer switching between sites.

The solution for this relies on GTS offer.

Through GTS APIs you can:

- create several VMs to host your application instances
- define a L4 Load Balancer over several instances
- define a domain alias to reach your instances

- and secure them.

When using Docker platform, the on-boarding process will provide you with a rounting mesh to handle these issues for you. Please refer to SG Docker documentation and see our proposal for deployment.

API-386 manual **Use a standard Load Balancer for resiliency** API uses a standard Load Balancer solution for its resiliency (GTS-VIP, GTS-Fabio, Docker, ...)	Availability

Use standard Load Balancer for scalability¶

A L7 Load Balancer provides you several features for high availability:

- expose several APIs behind the same domain name (with different routes)
- have a architecture tolerent to loss of an instance (dynamically add/remove instance to an HTTP route)

Some well known software are NGinx, Fabio, LinkerD.

The recommended usage within SG (bare metal, VMs) is to use Fabio in conjonction with Consul as service registry. For Docker environment you have a routing mesh for the GTS Cloud Platform or you can use the cloud provider routing mesh.

API-387 manual **Use a dynamic Load Balancer when scalability is needed** Load Balancer is automatically aware of new instances (tell how: GTS Fabio, Docker,...) or say "No needs for scalability" in review comment	Availability
API-388 manual **Use a managed dynamic Load Balancer to handle performance peeks or quick increase** An official managed solution (like GTS Fabio) is used when needing dynamic load balancing	Availability

Protect the API with quota/throttling¶

APIs that wish to manage the request rate of clients must define a quota and use the 429 (Too Many Requests) response code, if the client exceeded the request rate (see RFC-6586).

A quota is usually a max number of requests within a given window of time for a certain client.

This is because you have to keep your API reliable as also as scalable. Even if it is difficult to maintain a trusted relationship between

Such feature is often the key one for API gateways. It may be implemented simply per API instance, or across all API instances with help of distributed cache.

API-389 manual **Block excessive calls per client_id** API blocks client_id usage when exceeding the agreed usage plan	Performance
API-390 manual **Block excessive calls per client_id and operation** API blocks client_id usage when exceeding the agreed usage plan specific per operation	Performance

You should also use headers to provide further details to the client with the 429 error:

- Return a **Retry-After** header indicating how long the client ought to wait before making a follow-up request. The Retry-After header can contain a HTTP date value to retry after or the number of seconds to delay. Either is acceptable but APIs should prefer to use a delay in seconds.
- Return a trio of **X-RateLimit** headers. These headers (described below) allow a server to express a service level in the form of a number of allowing requests within a given window of time and when the window is reset.

The X-RateLimit headers (Github's proposal) are:

- X-RateLimit-Limit: The maximum number of requests that the client is allowed to make in this window.
- X-RateLimit-Remaining: The number of requests allowed in the current window.
- X-RateLimit-Reset: The relative time in seconds when the rate limit window will be reset.

Incorrect

```
openapi: 3.0.1
paths:
  /articles:
    get:
      responses:
        429: {}                         => miss headers
```

Correct

```
openapi: 3.0.1
paths:
  /articles:
    get:
      responses:
        429:
          headers:
            Retry-After: {}

openapi: 3.0.1
paths:
  /articles:
    get:
      responses:
        429:
          headers:
            X-RateLimit-Limit: {}
            X-RateLimit-Remaining: {}
            X-RateLimit-Reset: {}
```

API-391 auto-swagger-check **Protect the API with quota/throttling**
Use 429 status code with rate limiting standard headers (API-242 must be GREEN, else this rule may stay Pending) Performance

Not yet automated rule ensure that rate limiting is specified for every operations.

Be fault tolerant¶

APIs consume resources such as databases or other APIs. Their state and availability might impact the capacity of your API to function properly.

These resource can be temporary unavailable for a lot of valid reasons or incidents. A robust application should try to support small failures by applying the retry pattern. You should also be wary of actual faults and try to act on it if possible. The Circuit breaker pattern can help you achieve this.

Here are some tools/known libraries:

- specific policy in dotnet standard
- Netflix Hystrix which is now in maintenance mode
- Resilience4j inspired from Hystrix

API-392 manual **Be fault tolerant with your dependencies**
API implements retry and circuit breaker logic to mitigate dependencies unavailability (when eligible) Availability

Avoid throwing internal errors

Internal errors are proof that something went wrong on the API side and that the consumer cannot do anything with it. These should be treated as a production incident.

Therefore you should monitor them closely and have action plans to reduce and avoid them.

Expose release operating model¶

Releases are specific events that can impact the service provided to your clients. It is important to describe how it is performed and the impact it has:

- frequency
- duration
- downtime
- impact on performance
- notification/communication to consumers

The objective for APIs is usually to be able to deliver them continuously without impacting the service. Several patterns exist but all come at a cost.

API-394 manual **Do not fail your consumers during updates**
Consumers are always able to use the API during its official availability time, even if an update is in progress — Availability

Test, packaging, build, and release process are automated¶

There is a great deal of room for human error in API ecosystem, especially where deployment practices are concerned, and **most outages in large-scale production systems are caused by bad deployments**. Consider the organizational sprawl that accompanies the adoption of API architecture and what it entails for the deployment process: you have, at the very least, dozens (if not hundreds or thousands) of independent, isolated teams who are deploying changes to their APIs on their own schedules, and often without cross-team coordination between clients and dependencies. If something goes wrong, if a bug is introduced into production, or if a service is temporarily unavailable during deployment, then the entire ecosystem can be negatively affected.

To ensure that things go wrong with less frequency, and that any failures can be caught before being rolled out to all production servers, the API must be built, tested, packaged, and released through an automated pipeline, standardized across the owning entity. The whole process must be triggered with push-button human interactions only, removing completely any manual step.

API-500 manual **Test, packaging, build, and release process are automated** — Stability and Reliability
All these steps rely on automated tools

It is possible to release a new version of the API without service interruption¶

For example, you should have **at least 2 running instances of your API behind a load-balancer** and when you release a new version of your API, you **upgrade one instance first** (kill an old instance in version N-1 and start a new instance in version N) **then upgrade the second instance** so that at any time there is a running instance of your API even during a release (so without a service interruption). Of course, **versions N and N-1 must be backward compatible otherwise the breaking change would cause the failure of existing consumers**.

API-501 manual **It is possible to release a new version of the API without service interruption** — Stability and Reliability
Standard deliveries occur with no service interruption perceived by users

There is a canary phase in the deployment pipeline¶

Definition of a **Canary Release**: https://martinfowler.com/bliki/CanaryRelease.html

Once a new release has successfully been deployed to production, canary tests are run to validate the health of the release on the real production environment. For example, you could first route only 5-10% of the requests to the canary version and check that everything is ok. Then, route 50% of the requests to the canary version and check again. If everything is ok, then route all the requests to the canary version. Such deployment strategy is useful to limit the impact of a faulty new deployed version. Of course, **versions N and N-1 must be backward compatible otherwise the breaking change would cause the failure of existing consumers**.

API-502 manual **There is a canary phase in the deployment pipeline** Canary tests validate the production after delivery	Stability and Reliability

The Service Level Objectives of dependencies are compatible with the SLO of your API¶

The adoption of API architecture is sometimes driven by the idea that APIs can be built and run in isolation, as fully independent and replaceable components of a larger system. This is true in principle, but in the real world, every API has dependencies, both upstream and downstream. Every API will receive requests from clients (other APIs) that are counting on the service to perform as expected and to live up to its SLOs, as well as downstream dependencies (other services) that it will depend on to get the job done.

A stable and reliable API needs to mitigate dependency failures. This can be accomplished by having backups, fallbacks, caching, and/or alternatives for each dependency just in case they fail.

Before dependency failures can be planned for and mitigated, the dependencies of an API must be known, documented, and tracked. Any dependency that could harm an API's SLO needs to be included in the architecture diagram and documentation of the API and should be included on the service's monitoring dashboard(s). In addition, all dependencies should be tracked by automatically creating dependency graphs for each service, which can be accomplished by implementing a distributed tracking system across all APIs in the organization.

API-503 manual **The Service Level Objectives of dependencies are compatible with the SLO of your API** Asses the SLOs of dependency services	Stability and Reliability

There are backups, alternatives, circuit-breakers*/fallbacks, or defensive caching for each dependency to mitigate dependency failures¶

Definition of a **Circuit Breaker**: https://martinfowler.com/bliki/CircuitBreaker.html

Set up backups, alternatives, fallbacks, or caching for each dependency. The right way to do this is completely dependent on the needs of the service. For example, if the functionality of a dependency can be filled by calling the endpoint of another service, then failure of the primary dependency should be handled by the API so that requests are sent to the alternative instead. If requests that need to be sent to the dependency can be held in a queue when the dependency is unavailable, then a queue should be implemented. Another way to handle dependency failures is to put caching for the dependency into place within the service: cache any relevant data so that any failures will be handled gracefully.

API-504 manual **Defensive protections to mitigate dependency failures** There are backups, alternatives, circuit-breakers*/fallbacks, or defensive caching for each dependency to mitigate dependency failures	Stability and Reliability

Health checks accurately reflect the health of the API (and those checks must be called by load-balancers to stop routing to unhealthy APIs)¶

The health of an API at both the host level and the service level as a whole should always be known. This means that **health checks should be running constantly so that a request is never sent to an unhealthy host or service**. Running health checks on a separate channel (not used for general API communication) is the easiest way to ensure that health checks aren't ever compromised by something like a clogged network. Hardcoding "200 OK" responses on a /health endpoint for health checks isn't ideal for every API either, though it may be sufficient for most. Hardcoded responses don't tell you much except that the API was started on the host semi-successfully: any /health endpoint of an API should give a useful, accurate response.

If an instance of a service on a host is unhealthy, the load balancers should no longer route traffic to it. If an API as a whole is unhealthy (with all health checks failing on either a certain percentage of hosts or all hosts in production), then traffic should no longer be routed to that particular API until the problems causing the health checks to fail are resolved.

However, health checks shouldn't be the only determining factor in whether or not a service is healthy. A large number of unhandled exceptions should also lead to a service being marked unhealthy, and circuit breakers should be put into place for these failures so that if a service experiences an abnormal amount of errors, no more requests will be sent to the service until the problem is resolved. **The key in stable and reliable routing and discovery is this: preserve the API ecosystem by preventing bad actors from serving production traffic and accepting requests from other APIs.**

API-505 manual **Health checks accurately reflect the health of the API**
Health checks must be called by load-balancers to stop routing to unhealthy APIs Stability and Reliability

Scalability and Performance¶

APIs must be able to scale with the entire system without suffering from performance problems.

Scalability and performance are uniquely intertwined because of the effects they have on the efficiency of each API and the ecosystem as a whole. In order to build a scalable application, we need to **design for concurrency and partitioning**: concurrency allows each task to be broken up into smaller pieces, while partitioning is essential for allowing these smaller pieces to be processed in parallel. So, **while scalability is related to how we divide and conquer the processing of tasks, performance is the measure of how efficiently the application processes those tasks**. In a growing, thriving API ecosystem, where traffic is increasing steadily, each API needs to be able to scale with the entire system without suffering from performance problems.

To ensure that our APIs are scalable and performant, we need to require several things of each API:

- We need to understand its **growth scale**, both quantitative and qualitative, so that we can prepare for expected growth.
- We need to use our hardware resources efficiently, be aware of **resource bottlenecks** and requirements, and do appropriate **capacity planning**.
- We need to ensure that an API's **dependencies will scale** with it.
- We need to **manage traffic** in a scalable and performant way.
- We need to handle and **process tasks** in a performant manner.
- Last but not least, we need to store data in a scalable way.

Be stateless¶

It is always recommended to make your API stateless in order to favor its scalability and reliability.

A stateless API does not require to retain session information or status about each communicating partner for the duration of multiple requests. So each request should be sufficient to be treated at any time of the day and on any running instance of your API. Of cause, any request should not be refused because of the “session expired” reason.

Such session-less design avoids session stickiness - which means forcing a client to communicate always with the same server instance - and reduces the need for distributed cache or other centralized fast access data model.

API-395 manual **Be Stateless within your API's logic (no user sessions)**
The logic in API component is stateless so any received request can be managed on any instance of the API Performance

Support collection operations¶

If you expose a high volume of data a global retrieving operation is painful for consumers. Yet sometimes consumers need to retrieve many data. In this case your API should support collection operations on such resources.

An API must not force the client to do as many calls as requested records (gatling).

You may encounter such RESTful pattern that recommend to first reply a list of ids, and then let the client call for the resource by each of these ids, but we do not recommend this (unless using efficient transport layer as HTTP2).

Doing many small requests is a burden for the network and usually leads to bad performance compared to collected data.

Example of providing a list of ids instead of single one:

```
GET /deals?ids=123,234,23124,24234,234324,3423434
200 OK
{
  items: [
    { dealId:324, ..all other fields of a deal.. },
    { dealId:452, ..all other fields of a deal.. },
    ...
  ]
}
```

API-396 manual **Allow to retrieve or operate multiples resources at once**
Parameter can contain a list of identifiers to not force consumers to do multiple unitary calls Performance

See also Use common parameters for paging , Use common parameters for fields selection

Note that even if GraphQL seems efficient from client app point of view, a dummy implementation of the resolvers on server side may lead to gatling of underlying APIs/DBs and very bad performances (remember Entity EJBs ...)

Be Responsive¶

APIs should support techniques for improving responsiveness on client needs.

The max delay of a synchronous response to an API request (before first output data is received) must not be greater than

- 3 seconds in front of a physical user;
- 30 seconds in most common situations;
- 120 seconds in some extreme cases.

Not only because some intermediate HTTP network components may have shorter timeout than your gateways (your API should not become the shortcoming), but also because maintaining an open HTTP request for a long duration increases IT risks on simple protocol such as HTTP.

If the response can not be given within this delay, for read operations, you can consider streaming the results if possible. This lets your client consume the data on the fly. Standard emerged on this pattern so use them.

If the response can not be given within this delay, for write operation, an asynchronous pattern should be considered as follows:

- the API responds directly some request id with a 202 (the request was accepted but has not been finished yet); response provides a request id;
- the client polls regularly for the response using the request id;
- or the client waits for an asynchronous notification via some way like a webhook or WebSockets.

In case of polling the recommanded way is to provide a dedicated endpoint for such tasks that return:

- status 200 if the request is successful;
- a body containing the status of the task and the id of new resources (if any and when available)

Another implementation possibility is to return 204 status if the task is not completed yet.

For resiliency, such request state should usually be managed at database level to handle API instance failure.

Reduce bandwidth usage when huge data¶

While JSON is the default format of values (Use JSON as default format), it is meant to handle human-sized data which are relatively small documents. So trying to put a big volume of direct JSON data may cause bad performances in serializers or deserializers, especially on client side.

Standard APIs should support techniques for reducing bandwidth based on client needs. This holds for APIs that (might) have high payloads.

If you have to (de)serialize a JSON value which is greater than 1MB (a volume considered very important) you should resolve the (de)serialization with these common techniques :

- paging : provide explicitly a sorted sub-set of the result, but beware on the consistency to not miss any changed data of next sub-set
- streaming : allow to start consuming the input/output data before the data is completely prepared, but beware on behaviour when error occurs during the flow
- download : return an url of the built result instead of the data directly, on which the client can poll

See also Use specific patterns on binary data

API-399 manual **Reduce bandwidth usage when handling huge data, as paging**
Operations handling huge data use technical mechanism to handle it gently
Performance

Use specific patterns on binary data¶

When the API needs to consume or produce binary data alongs some usual data fields, as when uploading or downloading docs (pdf, docx, ...) or images alongs some kind of JSON metadata, you must be careful on the way to do it.

1. When volume is low (less than 10KB) you may embed it in your JSON object by using base64 encoded string (swagger type=string format=byte).

But bigger content should be handled another way else it may cause various issues on the JSON serialization.

2. You can use multipart content type, so one part is JSON (application/json) and other part is the binary content (application/octet-stream)
3. On uploads, you can put the data fields as query string parameters if not too many.
4. You can handle it in two steps calls

Ex: on download, you return first resource meta data and some download id

```
GET /documents/12345
{ documentId:12345, contentId:"file_12345.pdf", created:"2016-10-05T12:34:23Z"}
```

Then let the client download the binary data:

```
GET /contents/file_12345.pdf
  <PDF content>
```

Ex: on upload, client first pushes the json data, retrieving some id:

```
POST /documents
  { user:"john", type:"pdf" }
200
  { contentId:12345, ...}
```

then the client pushes the binary content using the provided id:

```
PUT /contents/12345
  <PDF content>
```

API-400 manual **Use specific patterns when handling binary data, dot not embed within JSON** Performance
Use one of the referenced patterns to manage binary data (if any)

Your API's growth scale is known (nb of users, nb of clients, nb of products, nb of deals...)¶

Definition of a **Growth Scale**: the key metrics that most influence the need for this API to scale.

Knowing the Growth Scale Determining how an API scales (at a very high level) is the first step toward understanding how to build and maintain a scalable API. There are two aspects to knowing the growth scale of an API, and they both play important roles in understanding and planning for the scalability of a service. The first is the **qualitative growth scale**, which comes from understanding where the service fits into the overall API ecosystem and which key high-level business metrics it will be affected by. The second is the **quantitative growth scale**, which is, as its name suggests, a well-defined, measurable, and quantitative understanding of how much traffic an API can handle.

Qualitative growth scales allow the scalability of a service to tie in with higher-level business metrics: an API may, for example, scale with the number of users, with the number of people who open a phone application ("eyeballs"), or with the number of orders (for a food delivery service).

The second part of knowing the growth scale is determining its **quantitative aspects, which is where RPS/TPS (Request or Transactions Per Second) and similar metrics come into play**. To determine the quantitative growth scale, we need to approach our APIs with the qualitative growth scale in mind: the quantitative growth scale is defined by translating the qualitative growth scale into a measurable quantity. For example, if the qualitative growth scale of our API is measured in "eyeballs" (e.g., how many people open a phone application), and each "eyeball" results in two requests to our API and one database transaction, then our quantitative growth scale is measured in terms of requests and transactions, resulting in requests per second and transactions per second as the two key quantities determining our scalability.

API-510 manual **Your API's growth scale is known**
Check factors such as nb of users, nb of clients, nb of products, nb of deals — Scalability and Performance

The traffic capacity that one instance of the API can handle is known¶

The resource requirements of an API are the hardware resources the API needs in order to run properly, to process tasks efficiently, and to be scaled vertically and/or horizontally. The two most important and relevant hardware resources tend to be, unsurprisingly, CPU and RAM (in multithreaded environments, threads become the third important resource). Determining the resource requirements of an API then entails quantifying the CPU and RAM that *one instance* of the service needs in order to run. This is essential for resource abstraction, for resource allocation and distribution, and for determining the overall scalability and performance of the API.

The most effective and efficient way to scale our service is to scale it horizontally: if our traffic is about to increase, we want to add a few more hosts and deploy our service to those new hosts. In order for us to know how many hosts we need to add, we need to know what our service looks like running on only one host: how much traffic can it handle? how much CPU does it utilize? how much memory? Those numbers will tell us exactly what the resource requirements of our API are.

API-511 manual **The traffic capacity is known**
The traffic capacity that one instance of the API can handle is known — Scalability and Performance

The resource bottlenecks of your API are identified¶

We can discover and quantify the performance and scalability limitations of our APIs by identifying resource bottlenecks. **A resource bottleneck is anything inherent about the way the API utilizes its resources that limits its scalability.** This could be an infrastructure bottleneck or something within the architecture of the service that prevents it from being scalable. For example, the number of open database connections an API needs can be a bottleneck if it nears the connection limit of the database. Another example of a common resource bottleneck is when APIs need to be vertically scaled (rather than horizontally scaled, where more instances/hardware is added) when they experience an increase in traffic: if the only way to scale an API is to increase the resources of each instance (more CPU, more memory), then the two principles of scalability (concurrency and partitioning) are abandoned.

If your service can only be scaled to meet growing traffic by deploying it to machines with more CPU and memory, then you have a scalability bottleneck and need to refactor the API so that it can be scaled horizontally rather than vertically, using concurrency and partitioning as your guiding principle.

API-512 manual **The resource bottlenecks of your API are identified**
See documentation for details — Scalability and Performance

The resource bottlenecks of your API are mitigated¶

Once the resources bottlenecks have been identified (see the rule The resource bottlenecks of your API are identified), are they actually mitigated / resolved ?

API-513 manual **The resource bottlenecks of your API are mitigated**
See documentation for details — Scalability and Performance

Your API is horizontally scalable¶

If your service can only be scaled to meet growing traffic by deploying it to machines with more CPU and memory, then you have a scalability bottleneck and need to refactor the API so that it can be scaled horizontally rather than vertically, using concurrency and partitioning as your guiding principle.

The most effective and efficient way to scale our service is to scale it horizontally: if our traffic is about to increase, we want to add a few more hosts and deploy our service to those new hosts.

API-514 manual **Your API is horizontally scalable** This prevents scalability bottleneck associated with vertical scalability	Scalability and Performance

Your API is auto-scalable¶

Definition of **Auto-scalable**: instances of this API are automatically started & stopped based on real-time metrics.

Your API should be horizontally scalable (see the rule Your API is auto-scalable), but ideally it should be automatically scalable, which means that new nodes (VMs, pods...) should be automatically started to manage the load peak, based on realtime metrics. And those new nodes should be stopped once the peak is over. Auto-scalability is easy to achieve on the public cloud (for example using ElasticLoadBalancer from AWZ) or on Kubernetes based infrastructures (for example OpenShift or Kubernetes offers from cloud providers).

API-515 manual **Your API is auto-scalable** instances of this API are automatically started & stopped based on real-time metrics	Scalability and Performance

A capacity planning is performed on a scheduled basis¶

To ensure that APIs can scale properly when traffic increases, we can perform scheduled capacity planning. The principles of capacity planning are pretty straightforward: determine the hardware needs of each API in advance, build the needs into the budget, and make sure that the required hardware is reserved.

To determine the hardware needs of each service, we can use the growth scales (both quantitative and qualitative), key business metrics and traffic predictions, the known resource bottlenecks and requirements, and historical data about the API's traffic. This is where qualitative and quantitative growth scales come in especially handy, because they allow us to figure out precisely how the scalability behavior of our APIs relate to high-level business predictions.

For example, if we know that (1) our API scales with unique visitors to the overall product, (2) each unique visitor corresponds to a certain number of requests per second made to our API, and (3) that the company predicts that the product will receive 20,000 new unique visitors in the next quarter, then we'll know exactly what our capacity needs will be for the next quarter.

API-516 manual **A capacity planning is performed on a scheduled basis** See documentation for details	Scalability and Performance

The API's traffic patterns are well understood (when are there peaks of charge, when is there low activity...)¶

Understanding current traffic pattern helps when interacting with the service on the ground floor in a lot of really interesting ways. When traffic patterns are clearly identified, both in terms of the requests per second sent to the service over time and all key metrics, changes to the service, operational downtimes, and deployments can be scheduled to avoid peak traffic times, cutting down on possible future outages if a bug is deployed and on potential downtime if the API is restarted while experiencing peak traffic load.

Closely monitoring the traffic in light of the traffic patterns and tuning the monitoring thresholds carefully with the traffic patterns of the API in mind can help catch any issues and incidents quickly before they cause an outage or lead to decreased availability.

When we can predict future traffic growth and understand the current and past traffic patterns well enough to know how the patterns will change with expected growth, we can perform load testing on our services to make sure that they behave as we expect under heavier traffic loads.

API-517 manual **The API's traffic patterns are well understood**
When are there peaks of charge, when is there low activity... See documentation for details
Scalability and Performance

Scalability or performance limitations in the way the API processes tasks have been identified¶

Production-readiness standardization requires each API to be both scalable and performant, which means that APIs need to be able to handle and process a large number of tasks at the same time, handle and process those tasks efficiently, and be prepared for tasks and requests to increase in the future. With this in mind, development teams should be able to answer three basic questions about their APIs: how their API processes tasks, how efficiently their API processes those tasks, and how their API will perform as the number of requests scales.

To ensure scalability and performance, APIs need to process tasks efficiently. In order to do this, they need to have both concurrency and partitioning. Concurrency requires that the service can't have one single process that does all of the work: that process will pick up one task at a time, complete the steps in a specific order, and then move on to the next, which is a relatively inefficient way to process tasks. Instead of architecting our service to use a single process, we can introduce concurrency so that each task is broken up into smaller pieces.

Taking the smaller pieces of these tasks, we can process them more efficiently using partitioning, where each task is not only broken up into small pieces but can be processed in parallel. If we have a large number of small tasks, we can process then all at the same time by sending them to a set of workers that can process them in parallel. If we need to process more tasks, we can easily scale with the increased demand by adding additional workers to process the new tasks without affecting the efficiency of our system. Together, concurrency and partitioning help ensure that our API is optimized for both scalability and partitioning.

API-518 manual **Scalability or performance limitations in the way the API processes tasks have been identified**
See documentation for details
Scalability and Performance

For any request that lasts more than 10s, an asynchronous solution is used¶

The nominal response time for an API is dozens of milliseconds. When the response time for an API is > 1 second, it often means that you are not in the right use-case for an API. If you want to exchange a lot of data, file transfer is probably more relevant. If you still want to exchange a lot of data using an API, then you must implement pagination/cursoring. If the call to the API is not large (either the request or the response) but just takes a lot of time (for example to perform a complex calculation), then use **asynchronous solutions** (for example Message Oriented Middlewares, like RabbitMQ) to temporally decouple the request and the reply. But do not block on synchronous call for more than few seconds. **Most cloud providers do not accept requests that take more than 10 seconds** (they kill the connection if the request takes longer).

API-519 manual **For any request that lasts more than 10s, an asynchronous solution is used**
See documentation for details
Scalability and Performance

Fault-Tolerance¶

APIs must remain fully or partly operational in case of failure of one component, underlying infrastructure, or dependency. The organization must be designed to detect failures and react in accordance with the Service Level Objectives.

The first step involved in building a fault-tolerant, catastrophe-prepared API is to architect away single points of failure. There should never be one piece of the ecosystem whose failure can bring the entire system to a halt, nor should there be any individual piece within the architecture of an API that will bring the API down whenever it fails. Identifying these single points of failure, both within the API and at a layer of abstraction above it, can prevent the most glaring failures from occurring.

Identifying failure scenarios is the next step. Not every failure or catastrophe that befalls an API is a glaringly obvious single point of failure that can be architected away. Fault tolerance and catastrophe-preparedness require that an API withstand both internal failures (failures within the API itself) and external failures (failures within other layers of the ecosystem). From a host failure to the failure of an entire datacenter, from a database to a service's distributed task queue, the number of ways in which an API can be brought down by the failure of one or more of its parts is overwhelming, scaling with the complexity of both the API itself and the API ecosystem as a whole.

Once single points of failure have been architected away and most (if not all) failure scenarios have been identified, the **next step is to test for these failures to see whether or not the API can recover gracefully when these failures occur and determine whether or not it is resilient**. The resiliency of a service can be tested very thoroughly through code testing, load testing, and chaos testing. This step is crucial: in a complex API ecosystem, merely architecting away failure is not enough — even the best mitigation strategy can turn out to be completely useless when components begin to fail. The only way to build a truly fault-tolerant API is to push it to fail in production by actively, repeatedly, and randomly failing each component that could cause the system to break.

> **Warning**
>
> Chaos testing is an advanced technique and must be run in a production environment only if you ran chaos testing successfully in a homologation environment and if you are very confident that your tests will not cause major impact on the production. **If in doubt, don't do it!**

Not all failures can be predicted, so the last steps in building fault-tolerant, catastrophe-prepared APIs are organizational in nature. Failure detection and mitigation strategies need to be in place and should be standardized across each API team, and every new failure that a service experiences should be added to the resiliency testing suite to ensure it never happens again. API teams also need to be trained to handle failures appropriately: dealing with outages and incidents (regardless of severity) should be standardized across the engineering organization.

The most probable of the API's failure scenarios and possible catastrophes have been identified¶

If we know anything about complex systems and large-scale distributed system architecture, it's this: that the system will break in any way that it can be broken, and any failure that could possibly happen will almost assuredly happen at some point during the system's lifetime.

APIs are complex systems. They are part of large-scale distributed systems (API ecosystems) and are therefore no exception to this rule. Any possible failure and any possible catastrophe will almost assuredly happen at some point in between the time an API's request for comments (RFC) is written up and the time the API is being deprecated and decommissioned. Catastrophes happen all of the time: racks fail in datacenters, HVAC systems break, production databases are deleted by accident (yes, this happens more than most developers would like to admit), natural disasters wipe out entire datacenters. **Any failure that can happen will happen**: dependencies will fail, individual servers will fail, libraries will become corrupted or lost entirely, monitoring will fail, logs can and will be lost (seemingly vanishing into thin air).

Once we've identified, mitigated, and (if possible) architected away any glaringly obvious points of failures in our API's architecture, the next step is to **identify any other failure scenarios and potential catastrophes that could befall our API**. We can separate these types of failures and catastrophes into four main categories, organizing them using their place in the API ecosystem stack. The most common catastrophes and failure scenarios are hardware failures, infrastructure (communication-layer and application-platform-layer) failures, dependency failures, and internal failures.

API-520 manual **The most probable of the API's failure scenarios and possible catastrophes have been identified** Fault-Tolerance
See documentation for details

There are clearly defined and tested mitigation strategies in case of failures¶

Architecting away single points of failure and identifying possible failure scenarios and catastrophes isn't enough to ensure that APIs are fault tolerant and prepared for any catastrophe. In order to be truly fault tolerant, an API must be able to experience failures and recover from them gracefully without affecting their own availability, the availability of their clients, and the availability of the overall API ecosystem.

The single best way to ensure that an API is fault tolerant is to take all of the possible failure scenarios that it could be affected by, and then actively, repeatedly, and randomly push it to fail in production.

API-521 manual **There are clearly defined and tested mitigation strategies in case of failures** Fault-Tolerance
See documentation for details

Your API undergo regular, scheduled load testing¶

A production-ready API needs to be both scalable and performant. It needs to handle a large number of tasks or requests at the same time and handle them efficiently, and it also must be prepared for tasks or requests to increase in the future. **APIs that are unprepared for increases in traffic, tasks, or requests can experience severe outages when any of these gradually or suddenly increase.**

From the point of view of an API development team, we know that traffic to our API will mostly likely increase at some time in the future, and we might even know by exactly how much the traffic will increase. We want to be fully prepared for these increases in traffic so that we can avoid any potential problems and/or failures. In addition, we want to illuminate any possible scalability challenges and bottlenecks that we might not be aware of until our API is pushed to the very limits of its scalability. **To protect against any scalability-related incidents and outages, and to be fully prepared for future increases in traffic, we can test the scalability of our services using load testing.**

API-522 manual **Your API undergo regular, scheduled load testing** Fault-Tolerance
See documentation for details

All possible failure scenarios are implemented and tested¶

To test for all failure scenarios, to make sure that APIs can gracefully recover from any potential catastrophe, there's one additional type of resiliency testing that needs to be in place, and it's known (quite appropriately) as chaos testing. **In chaos testing, APIs are actively pushed to fail in production, because the only way to make sure that an API can survive a failure is to make it fail all of the time, and in every way possible.** That means that every failure scenario and potential catastrophe needs to be identified, and then is needs to be forced to happen in production.

> **Warning**
>
> Chaos testing is an advanced technique and must be run in a production environment only if you ran chaos testing successfully in a homologation environment and if you are very confident that your tests will not cause major impact on the production. **If in doubt, don't do it!**

Running scheduled and random tests of each failure scenario and potential catastrophe can help mimic the real world of complex system failures: developers will know that part of the system will be pushed to fail on a scheduled basis and will prepare for those scheduled chaos runs, and they'll also be caught off guard by randomly scheduled tests.

API-523 manual **All possible failure scenarios are implemented and tested**
See documentation for details
Fault-Tolerance

A postmortem is systematically written after incidents and outages occurred¶

Postmortems need to be written to analyze and understand an incident or outage, severe incidents and outages need to be shared and reviewed, and a list of action items needs to be put together so that the development team(s) can complete them in order for the affected API(s) to return to a production-ready state (action items can often be fit into postmortems).

The most important aspect of incident follow-up is the postmortem. In general, a postmortem is a detailed document that follows every single incident and/or outage and contains critical information about what happened, why it happened, and what could have been done to prevent it. Every postmortem should, at the very minimum, contain a summary of what happened, data about what happened (time to detection, time to mitigation, time to resolution, total downtime, number of affected users, any relevant graphs and charts, etc.), a detailed timeline, a comprehensive root-cause analysis, a summary of how the incident could have been prevented, ways that similar outages can be prevented in the future, and a list of action items that need to be completed in order to bring the service back to a production-ready state.

API-524 manual **A postmortem is systematically written after incidents and outages occurred**
See documentation for details
Fault-Tolerance

Monitoring¶

The operational state of APIs must be monitored so that consumption rate changes and failures can be fixed and anticipated.

The majority of outages in an API ecosystem are caused by bad deployments. **The second most common cause of outages is the lack of proper monitoring.** It's easy to see why this is the case. If the state of an API is unknown, if key metrics aren't tracked, then any precipitating failures will remain unknown until an actual outage occurs. By the time an API experiences an outage due to lack of monitoring, its availability has already been compromised. During these outages, the time to mitigation and time to repair are prolonged, pulling the availability of the API down even further: without easily accessible information about the API's key metrics, developers are often faced with a blank slate, unprepared to quickly resolve the issue.

This is why proper monitoring is essential: it provides the development team with all of the relevant information about the API. When an API is properly monitored, its state is never unknown. **Monitoring a production-ready API has four components**:

- The first is proper **logging of all relevant and important information**, which allows developers to understand the state of the API at any time in the present or in the past.
- The second is the use of well-designed **dashboards that accurately reflect the health of the API** and are organized in such a way that anyone at the company could view the dashboard and understand the health and status of the API without difficulty.
- The third component is actionable and effective **alerting on all key metrics**, a practice that makes it easy for developers to mitigate and resolve problems with the API before they cause outages.
- The final component is the implementation and practice of running a sustainable **on-call rotation** responsible for the monitoring of the API.

With effective logging, dashboards, alerting, and on-call rotation, the API's availability can be protected: failures and errors will be detected, mitigated, and resolved before they bring down any part of the API ecosystem.

Your API logs all important requests¶

Logging is the first component of production-ready monitoring. It begins and belongs in the codebase of each API, nestled deep within the code of each service, capturing all of the information necessary to describe the state of the API. In fact, describing the state of the API at any given time in the recent past is the ultimate goal of logging.

One of the benefits of API architecture is the freedom it gives developers to deploy new features and code changes frequently, and one of the consequences of this newfound developer freedom and increased development velocity is that the API is always changing. **In most cases, the service will not be the same service it was 12 hours ago, let alone several days ago, and reproducing any problems will be impossible. When faced with a problem, often the only way to determine the root cause of an incident or outage is to comb through the logs, discover the state of the API at the time of the outage, and figure out why the service failed in that state.** Logging needs to be such that developers can determine from the logs exactly what went wrong and where things fell apart.

Determining precisely what to log is specific to each API. The best guidance on determining what needs to be logged is, somewhat unfortunately, necessarily vague: log whatever information is essential to describing the state of the service at a given time. Luckily, we can narrow down which information is necessary by restricting our logging to whatever can be contained in the code of the service. Host-level and infrastructure-level information won't (and shouldn't) be logged by the application itself, but by services and tools running the application platform.

API-530 manual **Your API logs all important requests** Monitoring
See documentation for details

Your API logs all errors¶

All errors generated by your APIs are recorded with enough context to investigate the cause of errors. Similarly, all errors experienced during interactions with external dependencies are logged.

For efficient error analysis, an efficient log collection is important. The error logs are collected and stored so that they can survive the failure of an API component, and can be explored without physical connection on different elements of the API infrastructure.

API-531 manual **Your API logs all errors** Monitoring
See documentation for details

All key metrics are displayed on dashboard¶

Every API must have at least one dashboard where all key metrics (such as hardware utilization, database connections, availability, latency, responses, and the status of API endpoints) are collected and displayed. A dashboard is a graphical display that is updated in real time to reflect all the most important information about an API. Dashboards should be easily accessible, centralized, and standardized across the API ecosystem.

Dashboards should be easy to interpret so that an outsider can quickly determine the health of the API: anyone should be able to look at the dashboard and know immediately whether or not the API is working correctly. This requires striking a balance between overloading a viewer with information (which would render the dashboard effectively useless) and not displaying enough information (which would also make the dashboard useless): only the necessary minimum of information about key metrics should be displayed. A dashboard should also serve as an accurate reflection of the overall quality of monitoring of the entire API. **Any key metric that is**

alerted on should be included in the dashboard: the exclusion of any key metric in the dashboard will reflect poor monitoring of the service, while the inclusion of metrics that are not necessary will reflect a neglect of alerting (and, consequently, monitoring) best practices.

API-532 manual **All key metrics are displayed on dashboard**
See documentation for details
Monitoring

There is an alert for every key metric¶

The detection of failures, as well as the detection of changes within key metrics that could lead to a failure, is accomplished through alerting. To ensure this, all key metrics — host-level metrics, infrastructure metrics, and API-specific metrics — should be alerted on, with alerts set at various thresholds. **Effective and actionable alerting is essential to preserving the availability of an API and preventing downtime.**

Alerts must be set up for all key metrics. Any change in a key metric at the host level, infrastructure level, or API level that could lead to an outage, cause a spike in latency, or somehow harm the availability of the API should trigger an alert. Importantly, alerts should also be triggered whenever a key metric is not seen.

API-533 manual **There is an alert for every key metric**
See documentation for details
Monitoring

Alert thresholds are set appropriately so that alerts will fire before an outage occurs¶

All alerts should be useful: they should be defined by good, signal-providing thresholds. Three types of thresholds should be set for each key metric, and have both upper and lower bounds: normal, warning, and critical. **Normal thresholds** reflect the usual, appropriate upper and lower bounds of each key metric and shouldn't ever trigger an alert. **Warning thresholds** on each key metric will trigger alerts when there is a deviation from the norm that could lead to a problem with the API; warning thresholds should be set such that they will trigger alerts before any deviations from the norm cause an outage or otherwise negatively affect the API. **Critical thresholds** should be set based on which upper and lower bounds on key metrics actually cause an outage, cause latency to spike, or otherwise hurt an API's availability. In an ideal world, warning thresholds should trigger alerts that lead to quick detection, mitigation, and resolution before any critical thresholds are reached. **In each category, thresholds should be high enough to avoid noise, but low enough to catch any and all real problems with key metrics.**

All alerts need to be actionable. Nonactionable alerts are those that are triggered and then resolved (or ignored) by the developer(s) on call for the API because they are not important, not relevant, do not signify that anything is wrong with the API, or alert on a problem that cannot be resolved by the developer(s). Any alert that cannot be immediately acted on by the on-call developer(s) should be removed from the pool of alerts, reassigned to the relevant on-call rotation, or (if possible) changed so that it becomes actionable.

API-534 manual **Alert thresholds are set appropriately so that alerts will fire before an outage occurs**
See documentation for details
Monitoring

All incoming and outgoing traces to/from the API are sent to a central collector (e.g. Zipkin, Jaeger...)¶

There are 3 pillars for the observability of a system: logs, metrics, and traces.

A **trace** is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system. Traces are a representation of logs; the data structure of traces looks almost like that of an event log. A single trace can provide visibility into both the path traversed by a request as well as the structure of a request. The path of a request allows software engineers and SREs to understand the different services involved in the path of a request, and the structure of a request helps one understand the junctures and effects of asynchrony in the execution of a request.

A **span** is a set of annotations that correspond to a particular RPC. A span represents a logical unit of work that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships.

A trace is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans. At the highest level, a trace tells the story of a transaction or workflow as it propagates through a (potentially distributed) system. **Traces are built by collecting all spans that share a traceId.** The spans are then arranged in a tree based on span-Id and parent-Id thus providing an overview of the path a request takes through the system.

API-535 manual **All incoming and outgoing traces to/from the API are sent to a central collector** Monitoring
Such as Zipkin, Jaeger, SG Monitoring

There are step-by-step triage, mitigation, and resolution instructions for each alert in the on-call runbook¶

Once an alert has been triggered, it needs to be handled quickly and effectively. The root cause of the triggered alert should be mitigated and resolved.

Step-by-step instructions for each known alert must be written that detail how to triage, mitigate, and resolve each alert. These step-by-step alert instructions should live within an on-call runbook within the centralized documentation of each API, making them easily accessible to anyone who is on call for the API. Runbooks are crucial to the monitoring of an API: they allow any on-call developer to have step-by-step instructions on how to mitigate and resolve the root causes of each alert. Since each alert is tied to a deviation in a key metric, runbooks can be written so that they address each key metric, known causes of deviations from the norm, and how to go about debugging the problem.

Two types of on-call runbooks should be created. The first are **runbooks for host-level and infrastructure-level alerts** that should be shared between the whole engineering organization — these should be written for every key host-level and infrastructure-level metric. The second are on-call **runbooks for specific APIs that have step-by-step instructions regarding API-specific alerts triggered by changes in key metrics**; for example, a spike in latency should trigger an alert, and there should be step-by-step instructions in the on-call runbook that clearly document how to debug, mitigate, and resolve spikes in the API's latency.

API-536 manual **There are step-by-step triage, mitigation, and resolution instructions for each alert in the on-call runbook** Monitoring
And people know how to access it

Expose health check¶

You must have an available health check on your application. It is a basic information to help you find instances that require action. It lets your load balancer distribute work-load efficiently and it let your clients know if they can call your API.

The health check exposed to clients, discovery tools or orchestration tools should be minimum (*UP*/*DOWN*) but you can use a more detailled health check internally (for management).

API-370 manual **Expose Health check**
API instances expose a health check endpoint telling if it is in sufficient health to handle requests Monitoring

Log incoming API calls¶

Your API should try to log all the API calls including:

- the request (path, caller information, body if it’s not too huge) - the volume of calls may complicate this;
- the request body in case of abnormal behavior;
- the response (status code, error message, size) - the volume of calls may complicate this;
- the response body in case of error;

If the amount of calls is very important, you must at least log the usage statistics (by counts or aggregations) and sample them for future performance diagnostics.

This logging aims at:

- helping API manager and API team understand how its API is being used and how to optimize API usage (etc. favor most consumed operation)
- providing in case of error more details about what the error is, when and why the error occurs
- judging the performance and impact of your API at client level in order to adapt infrastructure and client’s usage plan by current and forecast needs

API-371 manual **Log incoming API calls**
API calls (or statistics) are traced on at least 7 last days for diagnostic & usage — Monitoring

Log incoming API calls into SG Monitoring¶

As for the location of logs, you may save them in local files in each instance of your API but you should to centralize them into a common log repository. For this you can use:

- SG Monitoring which relies on Zipkin for ‘basic’ call information; this is the default system you should use for this purpose.
- an ElasticSearch cluster for detailed data (payloads) and standard applicative logs. You can use our libraries to Log HTTP requests/responses
- local disk if you don’t use a platform like Docker (but centralized logs ease investigation as your application scale.

Remember that you must guarantee the reliability of your API so that such logging operations should never become a risk for the API operations (like disk full, no memory, cpu saturation, network issue etc…). Most specialized libraries will help you in that regard.

The log data is proposed to be registered as structured Key/Value pair with the capability to provide necessary information like :

- usage statistics per operation/client/user
- other business needed data from url or in/out bodies
- error diagnostic information: status code, error response content, exception message and stacktrace
- performance diagnostic information: duration

> **Important**
>
> For automated rule API-373 to properly detect your usage of SGAPI Stack on SG Monitoring, you must :
>
> - Use SG Monitoring for APIs (zipkin) with the most recent SGAPI Stack In Java you should use the latest available version matching your Spring Boot minor version or (3.4.2, 3.9.2 or >= 3.13.0)
> - Connect to SGMON with your own API’s credential = from the credential tab within your API Card in the API Catalog, not from a Client App
>   - As temporary solution, you may also well declare the apiCode and apiVersion fields in your SGMON/APIs stack config

API-372 manual **Log incoming calls to SG Monitoring** Calls you receive are logged into SG-Monitoring	Monitoring
API-373 auto-catalog-check **Use the SG API stack to push to SG Monitoring the logs of received calls** For proper detection : use latest SGAPI stack (ex: java 3.11) and connect to SGMON with a credential from your API card's credential tab (not from a client app)	Monitoring

Use SG Monitoring with API's client_id¶

The client id used to contact SG Monitoring should belong to the API or application sending the spans. See How to manage API credentials and How to manage app credentials.

API-380 auto-catalog-check **Use SG Monitoring with API's client_id** The client_id used to call SG Monitoring must be a client_id of your API in API Catalog, and not of some Client App	Monitoring

Log outgoing API calls into SG Monitoring¶

When your API acts as a client calling other APIs, you should also log yours calls (request and received response) the same way as you log received calls for the same reasons but from the client perspective.

The default system to store this information is SG Monitoring which can also propagate data into your team ElasticSearch in order to aggregate all your API data in one place and keep chain data available.

> ### Important
>
> For automated rule API-376 to properly detect your usage of SGAPI Stack on SG Monitoring, you must :
>
> - Use SG Monitoring for APIs (zipkin) with the most recent SGAPI Stack In Java you should use the latest available version matching your Spring Boot minor version or (3.4.2, 3.9.2 or >= 3.13.0)
> - Connect to SGMON with your own API's credential = from the credential tab within your API Card in the API Catalog, not from a Client App
>   - As temporary solution, you may also well declare the apiCode and apiVersion fields in your SGMON/APIs stack config

≡

Calls from your API to other APIs are also traced and stored several days (native when using SG-Monitoring)	Monitoring
API-375 manual **Log outgoiing calls to SG Monitoring** Calls from your API to other APIs are logged into SG-Monitoring	Monitoring
API-376 auto-catalog-check **Use the SG API stack to push to SG Monitoring the logs of your sent calls to other APIs** For proper detection : use latest SGAPI stack (ex: java 3.11) and connect to SGMON with a credential from your API card's credential tab (not from a client app)	Monitoring

Correlate calls within an API chain¶

SG Monitoring services and the underlying technology help you track who is calling your APIs and how. The data sent to SG Monitoring is enhanced with a trace ID.

By propagating this contextual information you can greatly improve the diagnostic on a chain of applications.

This trace ID is in-band and should be propagated to each call of API chain in order to give a global overview of what happens within the API chain. So if your API is one of a chain of API calls, you must log and forward the trace ID that you received to next chained API.

By managing an out-of-band Span which contains the trace ID and sending it to SG Monitoring you will contribute the request as perceived by the client and the server.

If you use the provided libraries to interact with SG Monitoring then this feature is provided and you only need to activate it via configuration.

Note: The trace ID and span are required by SG Monitoring which is natively based on framework Zipkin and Open Traceing.

You may see the tutorial of SG Monitoring to know more about trace and span.

API-377 manual **Feed chain information to SG Monitoring** Incoming trace id is propagated to underlying called APIs	Monitoring

Do not log sensitive data of user¶

The EU General Data Protection Regulation (GDPR) and other similar regulations impose the **strict control** of user’s personal data and user’s activities in information systems as well as the **capacity to remove such information if user asks**.

If your API save user’s personal data into logs, it will be difficult to remove such data if user asks especially when the logs are saved in file. You have to also take enough care of your logs in order to avoid data leak or unknown data collect.

The best way to avoid violation of these regulations is not to save such information into your logs. That includes user’s email, last name/first name, windows login, sesame login, outlook name, ... File logs or elasticsearch data base or any other ways to persist log information (including db audit tables) whatever the user is SG internal or external.

> **Hint**
>
> If you have a requirement to know which exact user is, you may store and only store the opaque **sgconnect_id** which can be served to retrieve the client identity via SG Connect API but Do not log client identity details. The identifier of the user is not the oauth2 token but the field **sgconnect_id** received from the User Info. When using SGM API stack this information is available in the user claims.

API-378 manual **Do not log clear client app identity details if sensitive** When the identity of your internet client/partner is a C2 information, it is not directly visible in SG Monitoring data	Monitoring

Do not log client identity details¶

As the only information about which exact user is in your logs is the opaque **sgconnect_id**, you must not save any other details into your logs. Such details include the identity of client application/program/organization and so on.

This is because of the similar reason as GDPR for user, some regulations require not to keep trace or require easily removing activities of client or partner company (aka: application) in information systems.

The patterns for client applications are the same as for users whether they are internal or external.

See also Do not log sensitive data of user

API-379 manual **Do not log sensitive data of user** When the identity or information about your internet end user is a C2 information, it is not directly visible in SG Monitoring data	Monitoring

Credit: part of this page is copied from book “Production Ready Microservices” by Susan Fowler.