Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"

Network Technologies Department of the Faculty of Informatics

Development of a methodology for using microservice architecture in the construction of information systems

Master thesis
in specialty "Computer Science and Information Technologies" 112

Master's work supervisor
senior lecturer, candidate of technical science
Cherkasov D. I.

_____
(signature)

"____" _____ 2021 yr.

Made by student
Zhylenko O.V.

"____" _____ 2021 yr.

Kyiv 2021

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"

Department of Computer Science of the Faculty of Informatics

APPROVED

Head of the Department of Informatics

PhD, associate professor

_____ S.S. Gorokhovsky

(signature)

"____" _____ 2020 yr.

INDIVIDUAL TASK
for master's work

For the student of the Faculty of Informatics of 2 course of studying
THEME Development of a methodology for using microservice architecture in the construction of information systems

Output data:

Content of master thesis:
Individual task
Calendar plan
Abstract
Introduction
Part 1: Microservice architecture overview
Part 2: Requirements to microservice architecture
Part 3: Developing methodology for writing application using microservice architecture
Results
References
Applications

Issue date "____" _____ 2020 yr. Supervisor _____
(signature)

Task received _____
(signature)

Theme: Development of a methodology for using microservice architecture in the construction of information systems

Calendar plan of coursework execution:

№	Stage name	Deadline	Notes
1.	Getting of master's work topic	30.09.2020	
2.	Searching of appropriate literature	05.10.2020	
3.	Reviewing materials and building the structure of master's work	10.12.2020	
4.	Reviewing microservice architecture and writing first part	10.01.2021	
5.	Researching in requirements for microservice architecture	23.02.2021	
6.	Writing second part	15.03.2021	
7.	Developing methodology for writing services using microservice architecture	30.03.2021	
8.	Writing third part	15.04.2021	
9.	Writing master's results	20.05.2021	
10.	Master's work analysis with the supervisor	22.05.2021	
11.	Master's work changing according to the supervisor's remarks	23.05.2021	
12.	Creating of presentation	01.06.2021	
13.	Defending of the master's work	17.06.2021	

Student Zhylenko O.V.

Supervisor Cherkasov D. I.

“____” _____

Contents

Abstract

In this work will be defined what is microservice architecture, the most important quality attributes and system level requirements. We will gather guidelines grouped by quality attributes that should be used to reduce in future total cost of ownership system under develop. Created methodology will be used in synthetic Java project to demonstrate it on real example.

Key words: Microservice, Total cost of ownership, Quality attribute, Observability, Portability, Security, Maintainability.

Introduction

The concept of Total Cost of Ownership is universal, widely used, and not new. Commonly TCO is calculated for physical assets but it is not so popular for intellectual like informational systems. Usually we consider this concept afterwards when system already in production and costs of ownerships become too expensive. Systems cannot be controlled by engineers in effective way, not enough information is collected, root cause analysis takes too much time, etc. Issues in production and delays resolving that can cause to huge reputational damage for company and even to penalty fees if issue violate some constraints established by regulator on market.

Solving such types of problems requires a lot of efforts and can’t be done in short terms. What attributes must have the system to reduce risk of getting these problems in future? What system requirements system should satisfy? Defining of system level requirements is most effective on early stages in the system's design phase. Also, nowadays It is really difficult to imagine enterprise system that consist of only one deployment artifact. Great example is microservices architecture which is extremely popular now. Distributed systems add new challenges and extra complexity to implementation which will satisfy system requirements.

There are a lot of sources which give recommendation how to divide system to services based on business capabilities. Even more can be found which give general principles how to build microservices from technical point of view. Most of them ether too general to start using them in real world without significant adaptation ether cover only certain aspect without referencing to system level requirements. Great example of guidelines is The Twelve Factor App (8). It is very popular now but it does not provide concrete example of applying each point for real system.

The objective of this paper is to define quality attributes and requirements to build a system based on microservice architecture and develop methodology which consist of technical guidelines.

Object of the study is microservices quality attributes, system level requirements and best practices for building systems based on microservice architecture.

The research methods include analyzing of available articles, papers, ongoing researches and development of a guidelines for building distributed systems.

1. Microservice architecture overview

1.1. Microservice architecture definition

The microservice is a software development technique, which is a variant of the service-oriented architectural style or SOA (Service oriented architecture). Unfortunately, there isn't any unified definition for microservices. Martin Fowler said next: "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API." [4]

Usually when we are talking about microservices we mean self-contained pieces of business functionality which has clear interfaces. Common aspect of microservices is they tend to be built as cloud-native application. Microservice application has structure as a collection of loosely coupled services which are built around business capabilities. Each service is deployed independently and we tend to reduce centralize management for services. Other interesting aspect is each service can be written in different programming language and use different storage type.

To deep dive into microservice architecture definition it makes sense to compare it with the monolithic architecture which was standard architectural style during long year. Each application which is built using monolithic style is as a single unit. Often this unit consist of three main layers: representation layer (static content like HTML pages and javascript files) a database (usually this is relational database management system like Oracle), and business layer which is represented as a server-side application. The server-side application handle HTTP requests, perform domain logic, manipulate with data from the data storage, decide what HTML view should be presented to the client and return it. This server-side application is a *monolith*. If we want to introduce any changes to the system, we have to deploy new version of *monolith*.

All logic that handles requests runs in single process. This logic is divided by classes, functions, namespaces, etc. It depends on language that is used to build the

application. Running monolithic application on developers' local machines requires a lot of efforts in most cases. Monolith can be scaled horizontally behind load balancer. To release such application to production deployment pipeline is used.

When people start to deploy more application to the cloud, they start feeling deeply drawback of such architecture. Even small changes require to rebuild and deploy whole monolith. Also, it is difficult to keep good modular structure and loose coupling between modules. It's impossible to scale some feature independently.

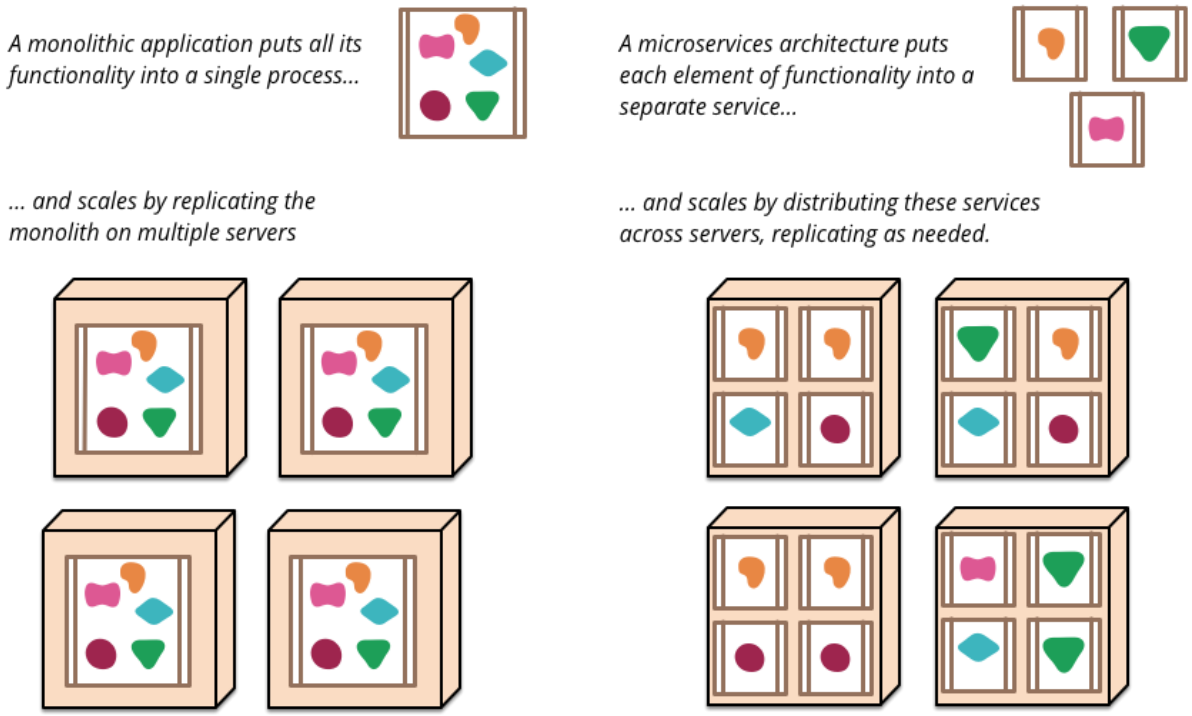

Figure 1.1.1. Microservices vs monolithic applications [4]

These led to the microservice architectural style. Developers start building applications as suite of services. Services can be deployed and scaled independently. There is a clear module boundaries and services can be written even in different programming languages. Also, each service can be managed by different team.

1.2. The twelve-factor app

The well-known "twelve factor application" methodology was drafted by developers Heroku and was presented by Adam Wiggins in 2011. The methodology is a set of rules for developing services that are widely used for launching, scaling and deploying applications. Since nowadays in most cases software delivered as services it became very popular.

According to official website the motivation of methodology is to raise awareness of some systemic problems we've seen in modern application development, to provide a shared vocabulary for discussing those problems, and to offer a set of broad conceptual solutions to those problems with accompanying terminology [8].

Let's deep dive in each point of methodology:

I. **Codebase.** One codebase tracked in revision control, many deploys.

All application code must be under version control. We should deploy artifact for development, testing, and production servers that are built from a single repository. Deployments may contain code in various branches that have not yet been added to the release. If you have several applications using common code, then the common code must be separated into a separate library and declared as a dependency. If you have several loosely coupled applications in one repository, then you must divide the application into several, each of which must be a twelve-factor application.

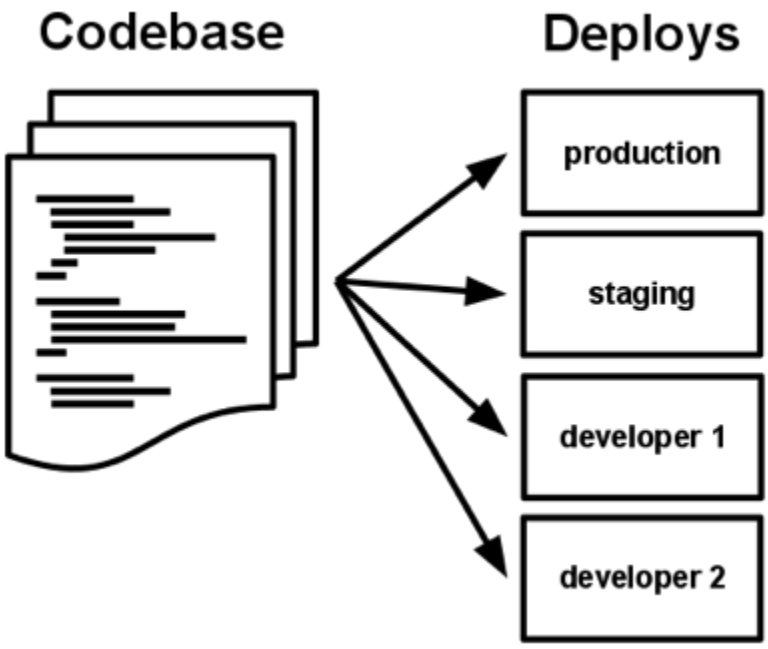

Figure 1.2.1. Single code base principle [22]

II. **Dependencies.** Explicitly declare and isolate dependencies

Applications should not have implicit dependencies. All dependencies, both system and libraries, must be written in the dependency manifest. All modern languages provide a package manager with a dependency manifest. For Java applications we have all dependencies in Maven *pom* file, for Node.js application – in *package.json*. In addition to the same application work on different platforms, this will allow new developers to join the project faster, or to connect new dependencies faster. Docker should be mentioned here, which isolates and explicitly declares even the dependencies of the OS.

III. **Config.** Store config in the environment

Configuration is all parameters that can be changed depending on where the application is started:

- logins, passwords and database addresses;
- third party services and API keys.

The code is environment independent as opposed to configuration. Therefore, the code should be stored in the repository, and the configuration in the environment. If you can grant public access to our code without compromising personal accounts, you are doing everything right. A fairly popular option is to use pre-built configuration sets (development, test and production). This separation is not a good solution, since adding new environments (QA, UAT, PERF) disproportionately increases the number of parameters that need to be maintained.

IV. Backing services. Treat backing services as attached resources

Third-party services are databases, mail services, caching servers, and APIs for various services. A twelve-factor application does not have to distinguish between local and remote services. Each service is a pluggable resource, the data for connecting to resources must be stored in the configuration. Replacing the local database with remote (Amazon RDS for example), or replacing the local mail server with a third-party one, should not affect the application code.

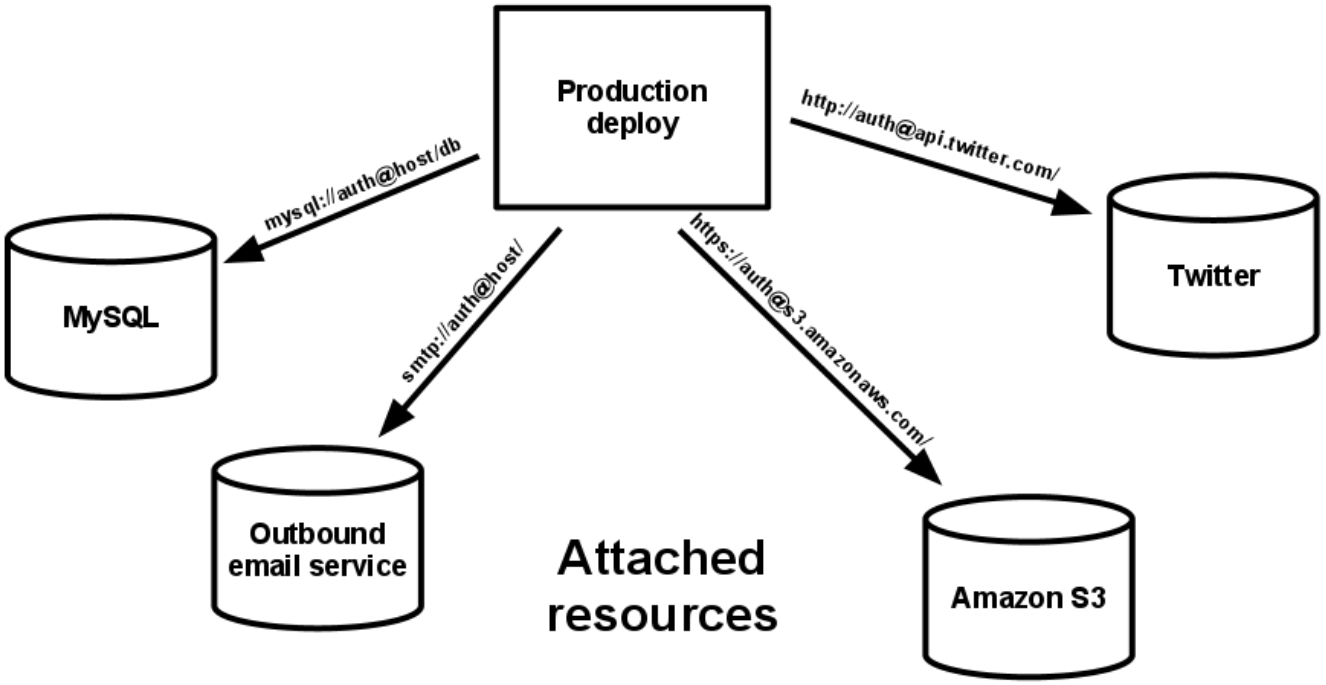

Figure 1.2.2. Backing service principle [22]

V. Build, release, run. Strictly separate build and run stages

Application deployment consists of three distinct phases:

- Build is the transformation of code into an executable package - downloading dependencies, compiling files and resources;
- Release - merging assembly with configuration. The release is immediately ready to run in runtime;
- Execution - launching a number of processes from the release.

When developing an application of twelve factors, these stages must be strictly separated. The build is initiated by the developer when he is ready to push the changes. This may be a longer process, but after that changes in the release code cannot be made. On the other hand, the execution phase should be as simple as possible so that it can be performed automatically in case errors, without developer involvement.

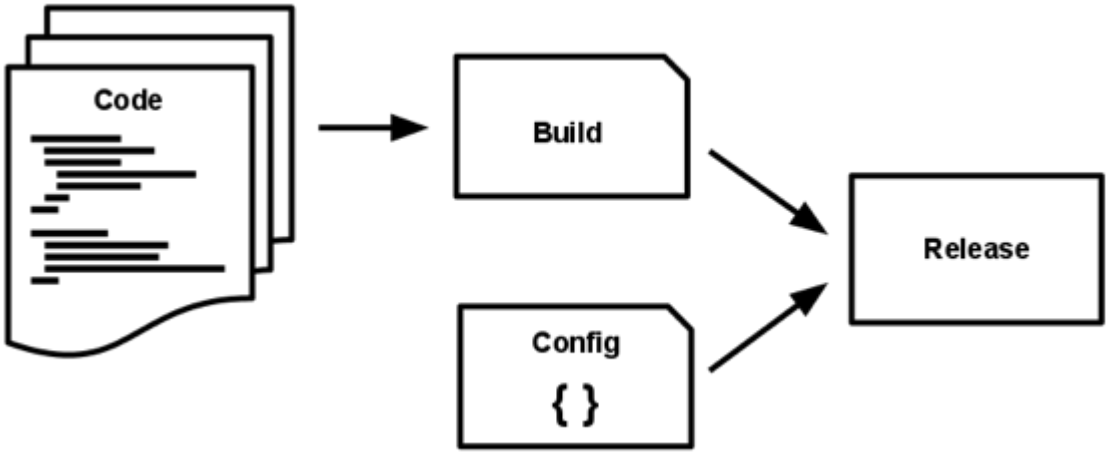

Figure 1.2.3. Build, release, run principle [22]

VI. Processes. Execute the app as one or more stateless processes

The application must run as one or more processes that do not store their internal state. An application can use data in RAM or on disk as temporary storage. For example, when transforming images. But any user data must be in persistent storage (pluggable resource).

First of all, this is related to session data, since the next user connection may occur to another process (due to a hardware failure, or a processor reboot).

VII. **Port binding.** Export services via port binding

Java application can be run inside Tomcat or Jetty but the 12-factor application is completely self-contained and does not rely on a web server. This is usually done with a dependency declaration to add a web server library to your application. Spring Boot, Quarkus, Micronaut, Play Framework comply with this principle.

VIII. **Concurrency.** Scale out via the process model

Different processes must be handled different tasks. For example, HTTP requests can be processed by a web process, while long-running background tasks can be processed by a worker process. This will allow in the future to scale only those processes that are necessary.

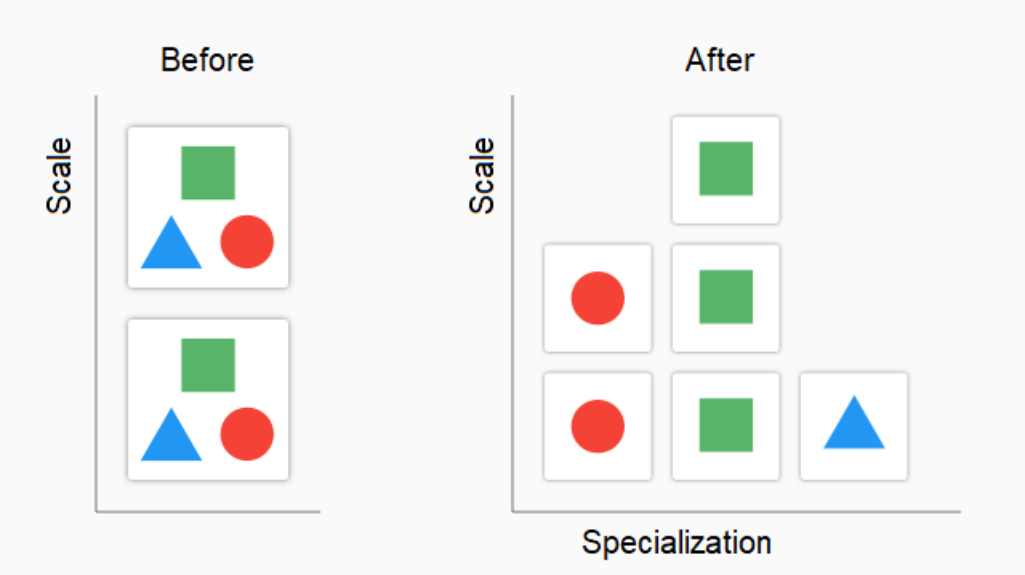

Figure 1.2.4. Process scaling by specialization

It is important that processes should not be run as daemon process. The process manager must monitor the application's output stream and respond to errors and process crashes.

IX. Disposability. Maximize robustness with fast startup and graceful shutdown

This is a logical consequence of processes that do not store their state. Maximize reliability with processes that start quickly and shut down gracefully, even if they crash. Background processes must return the current task to the queue, and processes that handle requests must complete requests correctly. Every task of any process must be available for re-execution (for example, using transactions).

X. **Dev/prod parity.** Keep development, staging, and production as similar as possible

There is a significant difference between development and deployment - it is done by different people, at different times, and with different tools. When introducing twelve factor application methodology, you should try to get development and deployment as similar as possible:

- The time gap - the code should reach the production version in a few hours after the developer wrote it;
- The personnel gap- the developer should be involved in the deployment of his code and monitor its work in progress;
- The tools gap - the development environment should match the testing environment (third-party services, OS) as much as possible.

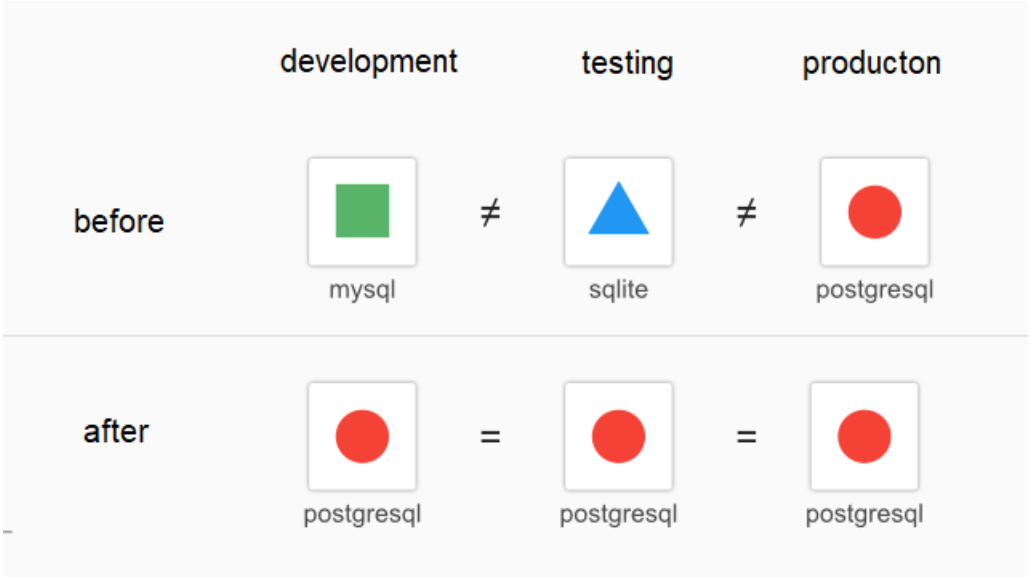

Figure 1.2.5. The tools gap

XI. **Logs.** Treat logs as event streams

A log is a stream of events and should be treated like a stream of events. An application should not write data to the log file itself, and manage files (archive or delete). The twelve

factor application outputs a log of its work to *stdout*. The manager who launches the process must direct the logic to the analysis or archiving system. This allows you to aggregate events from different processes, including both application processes and third-party services. The log analysis system allows you to analyze the current state of the entire system, as well as the analysis of previous work. In the developer's environment, the logs are simply viewed in the console.

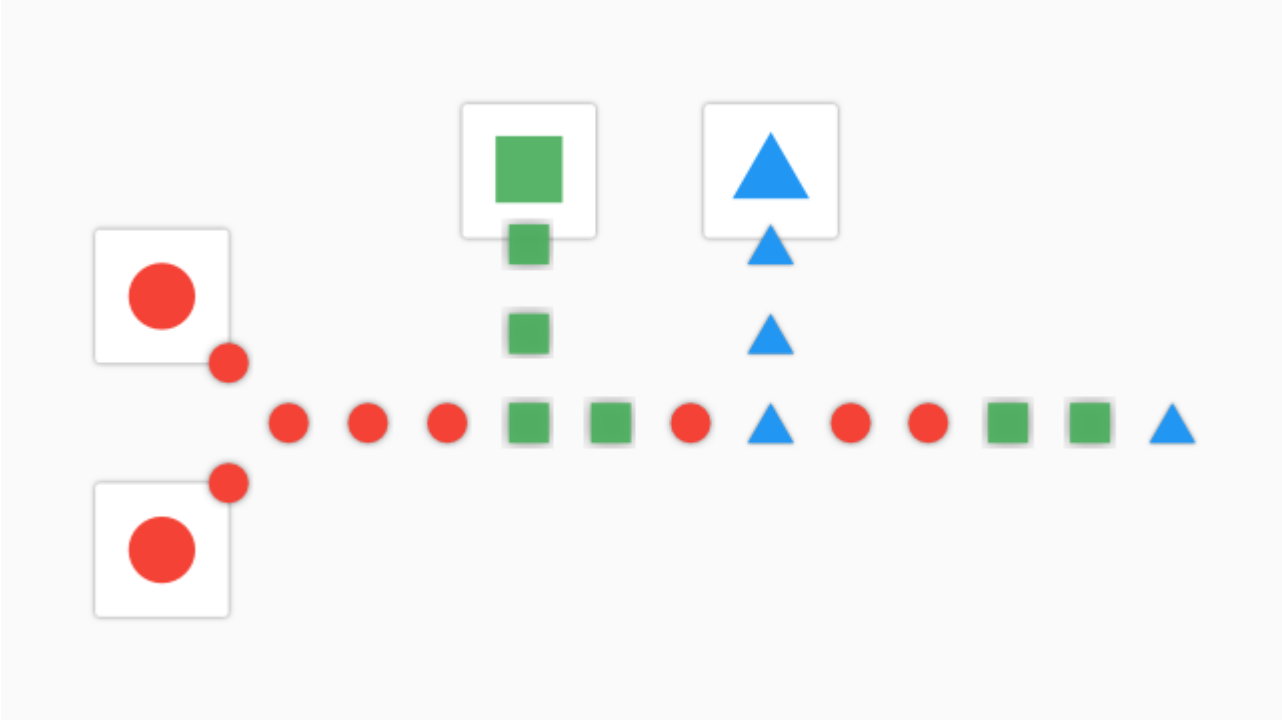

Figure 1.2.6. Logs stream

XII. Admin processes. Run admin/management tasks as one-off processes

One-time administration processes (migrations, database fixes) must follow the same rules as other processes:

- The task code must be in the repository to match the code of the main application;
- Dependencies must be declared in the main dependency manifest in order for the process to be executed in normal deployment;
- The configuration of the task must be in environment variables so that it can be executed in different environments.

2. Requirements to microservice architecture

2.1. Requirements definition

The requirement is a usable representation of the need, which helps to determine what kind of value should be delivered to fulfill it. There is multiple classification of requirements. For example, BABOK has described 4 types of requirements:

- Business Requirements
- Stakeholders Requirements
- Solution Requirements
- Transition Requirements

As per BABOK guide, the business requirement is defined as: Statements of goals, objectives, and outcomes that describe why a change has been initiated. They can apply to the whole of an enterprise, a business area, or a specific initiative. [6]

Stakeholder requirements as per BABOK guide: Describe the needs of stakeholders that must be met in order to achieve the business requirements.[6]

Solution requirements as per BABOK guide: Describe the capabilities and qualities of a solution that meets the stakeholder requirements. They provide the appropriate level of detail to allow for the development and implementation of the solution.[6]

Transition Requirements as per BABOK: Describe the capabilities that the solution must have and the conditions the solution must meet to facilitate transition from the current state to the future state, but which are not needed once the change is complete. They are differentiated from other requirements types because they are of a temporary nature.[6]

Classification provided above more suitable for business analyst and that is why we will use simplified classification which is more accurate from technical point of view.

Let’s define following categories of requirements:

- Functional requirements
- Non-functional requirements
    - Quality attributes
    - Constraints

Functional requirements define the behavior of a system or its component. How to behave or react on stimulations at runtime and what the system must perform and how to typically described by functional requirements. They describe specific functionality that define what a system is supposed to do (calculations, technical details, data manipulation and processing are involved)

Quality attribute requirements are qualifications of functional requirements or of the overall product. They usually answer questions like ‘how fast the function must be performed’ or ‘what is the time to deploy product’. We can consider quality attributes as qualifications of functional requirement. SEI proposed next definition of quality attribute: “A quality attribute (QA) is a measurable or testable property of a system, that is used to indicate how well the system satisfies the needs of its stakeholders”.[7]

Constraint is a design decision taken with zero degree of freedom. This decision was already made and you cannot influence on it in order to change. A great example of constraints can be restriction to use specific language because client has competency, or specific cloud provider.

2.2. Quality attributes

Observability

Observability is a measure of how well internal states of a system can be inferred from information of its external outputs. Usually developers confuse monitoring and observability. Observability is a property of a system but monitoring is the process. During

monitoring process where we translate outputs that we receive from application and infrastructure. The outputs can be logs and metrics. The main goal of monitoring is collecting all information necessary to be able to provide meaningful actions. Monitoring will not be helpful if system and its components don't externalize their state adequately.

Portability

Portability is the ability to deploy a product in various environments in a predictable way. This quality attribute includes containerization, configuration, versioning. The default tool for centonization is Docker. Configuration and versioning are implemented in different way and may be various depends on standards in organization.

Security

Security is the ability of the application resist to incorrect or malicious behavior of clients. This quality attributes includes next main areas:

1. Authentication and authorization of clients.
2. Translation, interpretation and protection of data.
3. Dependency and configuration management.
4. Auditing, logging, monitoring.

There are several projects which provide list of top vulnerabilities. OWASP and CWE are most popular among them. Developers should consider these lists during system development. For example, OWASP Top 10 which represents a broad consensus about the most critical security risks to web applications (https://owasp.org/www-project-top-ten/), can be used in during application development.

Maintainability

Maintainability is the ability to change a product with a predictable effort. To control maintainability static code analysis tools can be used. However, these tools may

not provide enough checks to ensure maintainability, so additionally developers should follow the code review practice.

Here are some recommendations:

1. **Minimize source code**

Lombok library can be used to auto-generate getters, setter and constructors in Java applications.

2. **Prefer declarative configurations**

Use declarative clients instead of request builders to consume data from HTTP services.

3. **Prefer infrastructure solutions**

Configure a reverse proxy instead to enable CORS. In any case application framework should not be used for this purpose.

3. Developing methodology for writing application using microservice architecture

3.1. Observability guides

Correlations

Systems have a high-level task that corresponds to the customer operation. In most cases system process more than one such type of operation at same time which cause to mess during investigation. Furthermore, one user operation in distributed systems can be translated to several interactions between parts of the system across multiple location. All log messages will be gathered by centralized logging tool. To make it straightforward to investigate problem, log message must be group in next way that each group is related to particular user operation.

Correlation ID is used to uniquely identify each user invocation. Correlation ID must be generated for each service that has external interface. If we are talking about Web application it would be controller which handle HTTP incoming requests. It might be service which reads data from event bus or queue. Once correlation ID is generated it must be propagated to all downstream service, message brokers and event busses.

Monitoring

Metrics must be collected and aggregated in single place. Errors, utilization, throughput and latency are the most important metrics. Prometheus and Grafana quite popular in nowadays. First one is used to store metrics while the second one to visualize them.

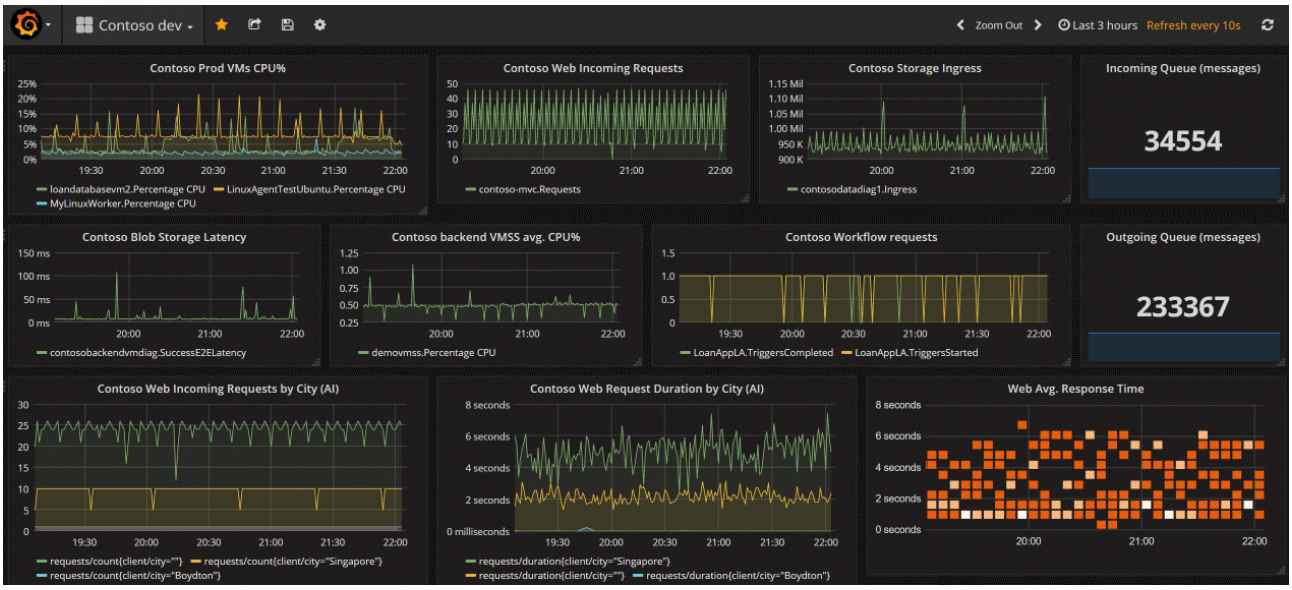

Figure 3.1.1. Grafana dashboard

Logging

Logs must be collected from every part of the system and stored outside node on which application is hosted. We should have single place to view, correlate logs and analyze them if needed. In microservice architecture we have distributed sources the number of which are increases during the time. Application should log events as they appear but this is not application responsibility to decide where to route logs. This is deployment configuration.

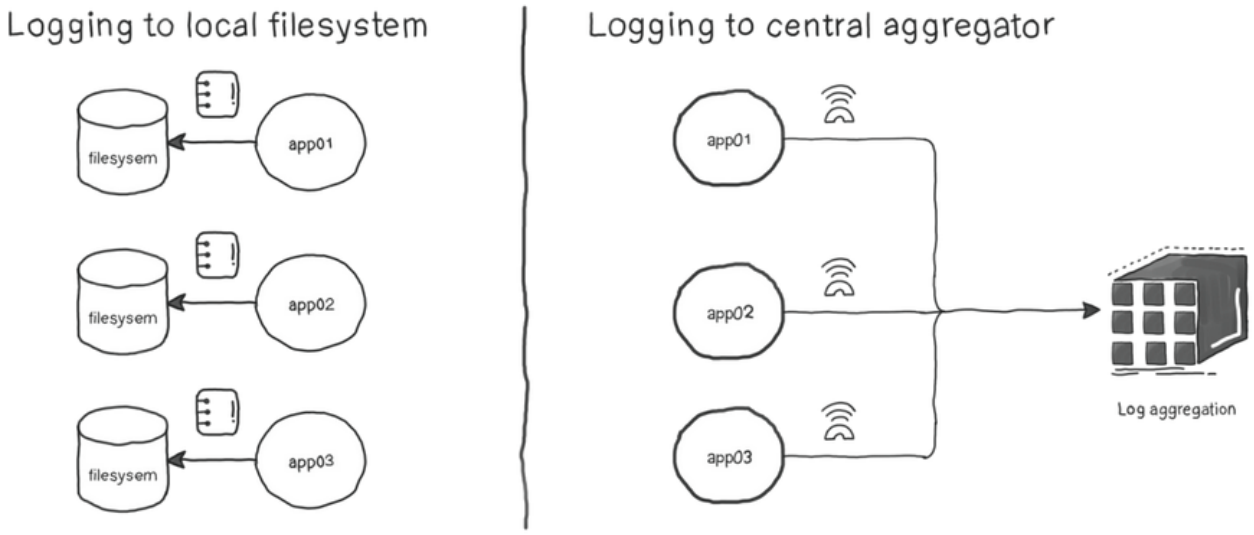

Figure 3.1.2. Types of logging [9]

But still we have to now where logs should go since this is environment configuration, we should use STDOUT and STERR as places to send log records. In service-oriented architecture it is common to have services written in different languages with different framework that has its own approach to redirecting logs. Since we are using STDOUT and STDERR it gives us guarantee of consistent way of collecting logs. Also, application can be written not only in different languages but also can be work on different OS. STDOUT and STDERR are inherent aspects of any OS by design so it provides consistence between all services.

Error handling

When HTTP server successfully receives clients request it must notify whether request was processed successfully or not.

In HTTP protocol we status codes which can be divided in 5 categories:

- **1xx (Informational)** – server acknowledges a request
- **2xx (Success)** – server completed the request successfully
- **3xx (Redirection)** – further actions are required from client in order to complete the request
- **4xx (Client error)** – client sent an invalid request
- **5xx (Server error)** – server failed to process a valid request due to an error on server side

Using the response code, a client can understand the result of a particular request and providing proper status code must be the very first in error handling mechanism.

Let's review some commonly used response codes

- 400 Bad Request – client sent an invalid request (required field is missing for example
- 401 Unauthorized – service failed to authenticate the client
- 403 Forbidden – client authenticated but does not have permission to access the requested resource
- 404 Not Found – the requested resource does not exist
- 412 Precondition Failed – some conditions in the request header fields was not satisfied
- 500 Internal Server Error – error on the server
- 503 Service Unavailable – the requested service is not available now

500 errors signal that some issues or exceptions happens on the server during request processing. This is internal error and it should not bother client. But we should try to minimize these types of responses to the client, so we should try to catch all such internal error and send corresponding status code wherever possible. For example, if

requested resource doesn't exist, we should return 404 instead of 500 status. It does not mean that 500 should not be used. It must be returned when unexpected situation happened like service outage.

Of course, just having the status code in most cases is not enough to understand the reason of problem so additionally more information must be provided. Developers start to add new fields to response body which cases to variety of error response so IETF devised RFC 7807[10] in order to generalize error-handling schema

Schema consist of five attributes [10]:

- "type" (string) - A URI reference [RFC3986] that identifies the problem type. This specification encourages that, when dereferenced, it provides human-readable documentation for the problem type (e.g., using HTML [W3C.REC-html5-20141028]). When this member is not present, its value is assumed to be "about:blank".
- "title" (string) - A short, human-readable summary of the problem type. It SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization (e.g., using proactive content negotiation; see [RFC7231], Section 3.4).
- "status" (number) - The HTTP status code ([RFC7231], Section 6) generated by the origin server for this occurrence of the problem.
- "detail" (string) - A human-readable explanation specific to this occurrence of the problem.
- "instance" (string) - A URI reference that identifies the specific occurrence of the problem. It may or may not yield further information if dereferenced.

Here is what provided in RFC 7807 document as response example:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
 "type": "https://example.com/probs/out-of-credit",
 "title": "You do not have enough credit.",
 "detail": "Your current balance is 30, but that costs 50.",
 "instance": "/account/12345/msgs/abc",
 "balance": 30,
 "accounts": ["/account/12345",
              "/account/67890"]
}
```

Comply to RFC 7807 is optional and is recommended in big organization where very difficult to agreed on standard error message schema across all units.

Let’s review error messages from such a giant company like Twitter and Facebook.

```
curl -X GET https://api.twitter.com/1.1/statuses/update.json?include_entities=true
```

Here is a Twitter API response:

```
{
    "errors": [
        {
            "code":215,
            "message":"Bad Authentication data."
        }
    ]
}
```

As you can see response contains list of errors but in our case, we have only error in list.

Let's call Facebook Graph API:

```
curl -X GET
https://graph.facebook.com/oauth/access_token?client_id=oleksii&client_secret=sc&gr
ant_type=some
```

Next error occurs:

```
{
    "error": {
        "message": "Missing redirect_uri parameter.",
        "type": "OAuthException",
        "code": 191,
        "fbtrace_id": "AWswcVwbcqagrHgjG80MtqJ"
    }
}
```

There is common *code* field in both responses. It is very convenient to have error codes. First of all, behind single HTTP Status code can be several reasons. Having additional business error identifier help to granularly distinguish it. Second point is having number error identifies simplify analyzing of problems for non-human clients.

Health checks

Our soft has dependencies on computer hardware, libraries which are maintained by other teams. None of them cannot provide 100% guarantees that everything will work perfect all the time. It’s impossible to build 100% reliable software on top of unreliable components because of that it’s obvious that service is going to fail sometime after release to production. Mechanism how to detect this failure before users do should be implemented. And if it does, you have to detect it somehow. We all agree that it's better to do it before end-users do.

Here are types of failures which can cause in Java application:

- **Bugs.** Can be introduced by developer during coding. There is some correlation between number of bugs and line of codes. Steve McConnell in his book said that industry average about 15 - 50 errors per 1000 lines of delivered code [20].

- **Memory Leaks.** When the garbage collector fails to recycle specific area of the heap memory leaks can occurs. This area grows over time. After some period of time out of memory can happen but until that number of garbage collector pauses will be increased significantly which reduce the performance of whole application.
- **Thread Leaks**. Unclosed resources or threads which are not managed properly, can cause to thread leaks. This will lead your JVM to a stall while the CPU is going to spin at 100%.
- **Configuration Issues.** Configuration issues are tricky since they can be caught in the same environment they are referring to. It means production configuration issues can be reproduced during production deployment only.
- **Deadlocks.** JVM do not offer deadlock detection. This means that threads hanging in deadlock will wait forever until application is stopped.
- **Connection Pool Misconfigurations.** Connection pool settings should be reviewed carefully, in other can there is a risk that connection pool will start causing failures.

Redundancy is the simplest way to introduce fault-tolerance into any system. Introducing redundancy for services means making the process redundant. Have multiple processes running at the same time. If one of them start behave in a wrong way then others can take the workload. Of course, some kind of coordination is required in other case it will not work. Usually, coordination is done by using health check mechanism.

The coordination can be a container orchestrator or a load balancer. The main role is to hide implementation details from the clients. Coordinator uses cluster of services and show them as a single logical unit. In order to do this the workload must be scheduled to those services only which are reported to be healthy. Coordinator asks each running process in the cluster about their health status. Using this information further actions can be taken. Coordination strategy based on health status information is not our topic so we list some most used of them:

- Restarts
- Alerting
- Traffic shaping
- Scaling
- Deployments

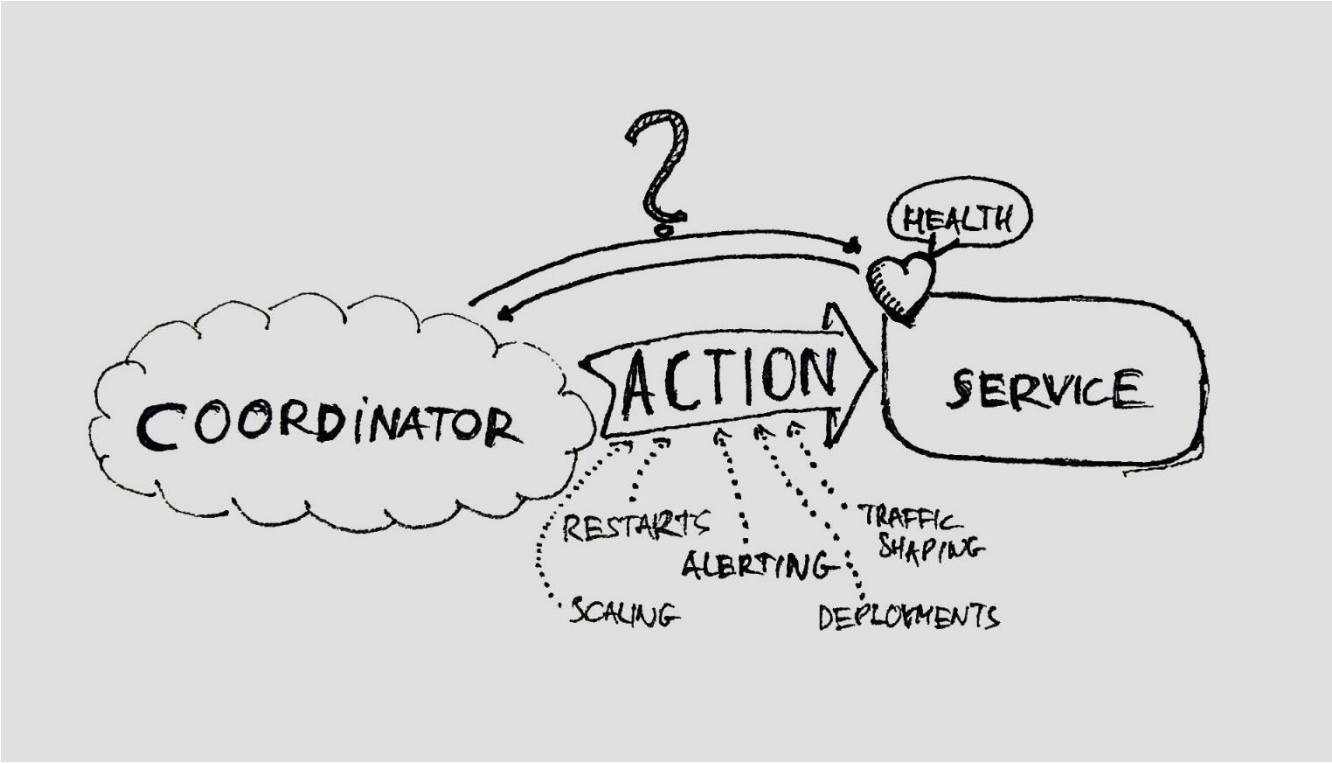

Figure 3.1.3. Coordinator actions [21]

How let’s review the ways how health checks can be implemented.

No health checks. It is the easiest option. In this case orchestrator can still restart container if it stopped but it cannot apply other strategies. Container orchestrator able to catch memory leaks if JVM exits with *OutOfMemoryError*. In case of a thread leak transactions will become slower until they completely stall. Alerts can be set up based on the type of HTTP responses if L7 load balancing is used but it is not possible to set alerts if TCP load balancer is used because of the lack of interpretation of the HTTP response codes.

Shallow health checks. In this implementation we just usually just verify if the HTTP pool is able to provide some kind of response. Static content or empty page with an HTTP 2xx response code is returned.

Here is simple health check response:

```
{
  "status": "UP"
}
```

If it falls under a predefined threshold, the service report itself as unhealthy. In Kubernetes, for example, we have can configure a *liveness/readiness* probe for containers. It will restart service automatically if it reports as unhealthy. In such case issues with the HTTP pool, like thread leaks and deadlocks can be recovered. Also fault that a more generic can be process (memory leaks). But issues that occurs further in the stack cannot be checked (trying to save data in database for example). In this scenario we will get response from health check but load balancer will not be able to handle this situation (Usually health checks are integrated with load balancers and load balancer route traffic to healthy services). There are 3 options how to deal with this situation:

- Try to store the request in the service and retry later. The request will be lost if coordinator restart service.
- Fail fast. Caller should retry. The problem is moved to one layer above.
- Include database in the health indicator. This led to deep health checks.

Deep health checks. Deep health checks include the surrounding of service. For example, Spring Boot has a lot of automatically configured health indicators in started Actuator. Here's an example of deep health check:

```
{
   "status":"DOWN",
   "details":{
      "serviceA":{
         "status":"UP",
         "details":{
            "Service A":"Available"
```

```
            }
         },
         "serviceB":{
            "status":"DOWN",
            "details":{
               "Service B":"Not Available"
            }
         },
         "db":{
            "status":"UP",
            "details":{
               "database":"H2",
               "hello":1
            }
         },
         "diskSpace":{
            "status":"UP",
            "details":{
               "total":250790436864,
               "free":36591120384,
               "threshold":10485760
            }
         }
      }
   }
```


In such implementation it makes sense to have multiple health check endpoints for the controlling logic (like in Kubernetes). Different actions can be applied for different failures. Liveness, readiness and startup probes can be good examples. The traffic will be forwarder to a service only if its dependencies are accessible. This all-or-nothing approach might be too restrictive if we are taking into account upstream dependencies with deep health checks.

Let’s consider situation when database dependency is not accessible. There might be two different root causes of this problem: some problem with connection pool (we reach limits) or some problems with database. In case of database issues other services will be unhealthy also and load balancer will remover every instance. In practice, it does not make sense to totally stop processing requests. This bug with health checks can cause outage of production. Load balancing settings should be revisited in a such way that upper limit of instances that can be removed is set.

Other important point is health check synchronization with circuit breaker configuration. If real service data on which we have dependency is not available and

circuit breaker return cached data should this service be considers as healthy or unhealthy? This deep health check limitation can be solved but unfortunately not easily. Synthetic request can be sent with some period of time. For example, we can query data from database. It is mor difficult with message queues and event buses because we have to send some message which can trigger corresponding activity on consumer side. Value in message must be distinguishable by consumers from real messages. Also, synthetic traffic must be filtered somehow in monitoring infrastructure but this produce more overhead than shallow health checks.

Passive health checks. The main idea of passive health checks is to not validate status of resources every time and mark resource as unhealthy when the failure rate is higher than the configured threshold. Service should not be removed forever, so time limit for the removal should be implemented. After a while, we can check services health again to see if the situation is getting better. This health check approach is very similar to circuit breaker pattern. I see only benefit of using passive health checks is that we do not need to synchronize the configuration with other fault-tolerant patterns (circuit breakers, fullbacks, etc.). It makes sense to this type of health checks when it is provided by framework or other tool like proxy, load balancer. For example, Envoy offers this functionality by default which is named as outlier detection.

Tracing

Trace must be propagated to all services, message brokers and event buses. This information must be collected on aggregation tool.

Let’s review why it is so important. When performance bottlenecks happen, developers spend a lot of time on monitoring and parsing logs. When logging the timings of individual operations into a log file, it is difficult to understand what is the cause of this operation or to track the sequence of actions or the time shift of one operation relative to another in different services.

Here are the problems which can be solved by tracing:

1. Find performance leaks within one service and across the entire execution tree between all participating services. For example:
    a. Lots of short sequential calls between services.
    b. Long waits for I/O operations, such as transferring data over the network or reading from disk.
    c. Long data parsing.
    d. Long operations requiring CPU.
    e. Portions of code that are not needed to get the final result and can be deleted, or run delayed.
2. Clearly understand in what sequence what is called and what happens when an operation is performed.

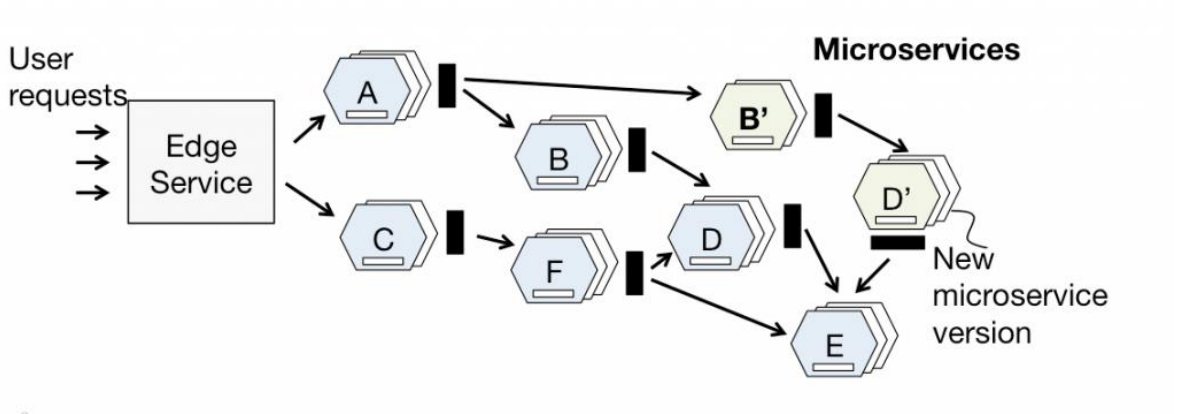

Figure TODO: 3.1.4. Microservices calls topology [24]

Without trace or detailed documentation for the entire process, it is very difficult to understand what is happening in microservice topology which is present in figure above.

3. Collecting information about the execution tree for analysis in future. At each stage of execution, additional information can be added to the trace that is

available at this stage and then figure out what input data led to such a scenario. For example:

a. User ID
b. Rights
c. Type of selected method
d. Log or execution error

4. Traces can be transformed into subset of metrics that can be analyzed in future.

3.2. Portability guidelines

Containerization

Using containers is production standard in nowadays and Docker is default tool for this purpose. However not always developers think how they build images. In simplest cases it can cause to not optimal image size and performance for allocated resources, long image build stage. But in worst case it can expose critical vulnerabilities.

Here are some recommendations that can help create optimal for production docker images.

Static layers should be at the top of changing layers. This will reduce build time. If changing layers are put above static layer in docker file if will cause to rebuilding all bottom layers.

Here is example of wrong docker file:

```
COPY lib/* /deployment/lib/
COPY sample-app.jar /deployment/

RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh
```

This file should be changed in next way:

```
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh

COPY lib/* /deployment/lib/
COPY sample-app.jar /deployment/
```

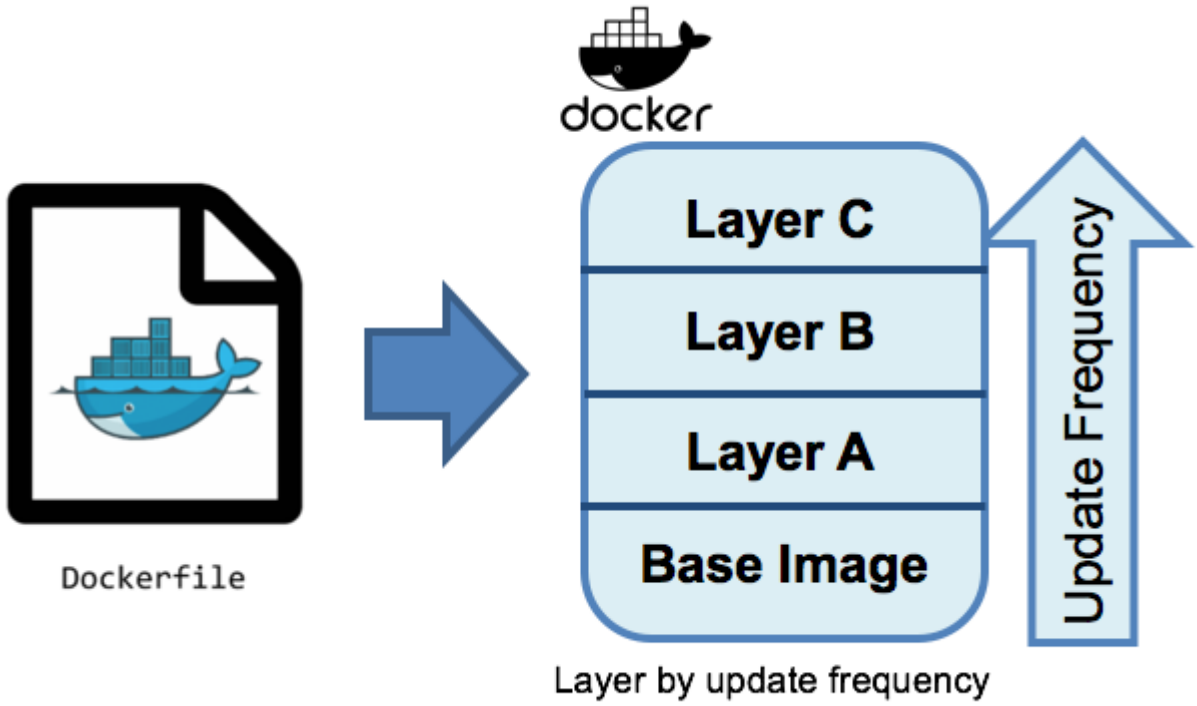

Figure 3.2.1. Docker layers by update frequency [16]

Try to use specific names when copying files. If wildcard is used in copy command can break cache if new file will be created in directory with the target file. This is not critical in some cases it's necessary to use wildcard but you have to be careful with them.

Initial docker file:

```
COPY *-app.jar /deployment/
```

How it can be changed:

```
COPY sample-app.jar /deployment/app.jar
```

Group layers. Try to group your commands into as much as possible. It reduces layers count, cache size and as a result images size.

Bad practice:

```
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh
```

Can be combined to:

```
RUN apt-get update \
 && apt-get -y install \
    openjdk-8-jdk ssh
```

Reduce image size use flags *--no-install-recommends* **for apt-get command.** It will not install unnecessary staff.

Before:

```
RUN apt-get -y install openjdk-8-jdk
```

After:

```
RUN apt-get -y install --no-install-recommends openjdk-8-jdk
```

Remove apt manager cache. Use `rm -rf /var/lib/apt/lists/*` command for *apt-get*.

Before:

```
RUN apt-get update \
 && apt-get -y install --no-install-recommends openjdk-8-jdk
```

After:

```
RUN apt-get update \
 && apt-get -y install --no-install-recommends openjdk-8-jdk
 && rm -rf /var/lib/apt/lists/*
```

Choose base image carefully. When we put our application in Docker container, we build on existing image on which we reference in *FROM* statement in Dockerfile. It is

absolutely essential to using well-maintained base image. We have to be sure that image was patched for security issues and does not contains malware software or scripts. Use certified images from Docker Hub or images from software vendor like Oracle, Red Hat and other if you or your company pay for support. During base image selection try to using a smaller base image as possible It will help keep the final image size small. For example, Alpine or Google's Distroless are small. But you have be aware that if you use Alpine Linux as base image, you're using *musl* and not *glibc* (the GNU C Library). This potentially might impact your application's performance and supportability. Google's distroless image use *glibc*. You can take a look at Project Portola. Official documentation states that goal of this Project is to provide a port of the JDK to the Alpine Linux distribution, and in particular the *musl* C library [13].

Other option is using minimal Linux base image like Alpine and then install all necessary tools by yourself (JRE, npm, etc..) if you prefer to not using community baes images. However, you have to plan this base image will be updated when some new toll version is released. It is critical because it will keep you up to date with security updates.

Be careful with automatic Docker image generation tools. There are a lott of great tools and plugins for build system which create Docker images and even able to publish them to registry. From a developer perspective, this looks convenient since they do not need to maintaining Dockerfiles. An example for Java application can be JIB Maven plugin. It must be configured and then can be called using *mvn jib:dockerBuild* command. Here is a basic JIB configuration:

```
<plugin>
   <groupId>com.google.cloud.tools</groupId>
   <artifactId>jib-maven-plugin</artifactId>
   <version>3.0.0</version>
   <configuration>
       <to>
           <image>order-service</image>
       </to>
   </configuration>
</plugin>
```

Something similar can be done with Spring Boot if you are using version 2.3 and up by calling next maven command:

```
mvn spring-boot:build-image
```

In both cases, these containers are relatively small. That is because they are using distroless images or build packs as a base image. But other important questions is whether these containers are safe? Deeper investigation is required but event even then we cannot be sure whether this status is maintained in the future. It does not mean that that these tools should not be used for Docker images creating. Investigation of container security aspects must be performed and scanning containers is a good start. Having full control of Docker image creating by managing our own Dockerfile in a proper way is more preferable approach.

Using multi-stage builds. Sometimes not needed to install all required staff on machine that using for building application distributing. In such case distribution can be build inside docker images. Since it produces a lot of files which are not needed for execution multi-stage builds are essential in such cases. Image is built using all the tools needed and second stage where we create the actual production image.

Here is example of multi-stage Dockerfile:

```
FROM maven:3.6.3-jdk-11-slim AS build
RUN mkdir /project
COPY . /project
WORKDIR /project
RUN mvn clean package -DskipTests


FROM adoptopenjdk/openjdk11:jre-11.0.9.1_1-alpine
RUN mkdir /app
COPY --from=build /project/target/order-service.jar /app/order-service.jar
WORKDIR /app
CMD "java" "-jar" "order-service.jar"
```

The crucial aspect of multi-stage build is preventing sensitive information leak. It is highly possible that connection to a private repository is required. In java application it

can be done in Maven *settings.xml* on local machine or build environment. When using multistage builds the *settings.xml* can be copied to building container. The settings with the credentials will not end up in final production image. Also, credentials as command line arguments can be used safely because it will not end up in the production image. Multiple stages can be created and only necessary results will be copied to final production image. This is not a great way to separate concerns but in a such way developers can be sure that data will not leak production environment.

Build one image for all environments. Docker containers are designed to run the same anywhere. This means one image is built once, and can be run in all environments (dev, qa, prod) by just changing config. Deploying one image into several target environments can be achiever using externalized configuration. One of the biggest container anti-pattern is building one image per environment. This idea is not new and based on the principle of building one binary for all environments, as this section from Continuous Delivery by Jez Humble and David Farley explains [14]: “The binaries that get deployed into production should be exactly the same as those that went through the acceptance test process… It must be possible to deploy these binaries to every environment. This forces you to separate code, which remains the same between environments, and configuration, which differs between environments.”

Containerization concept is about creating a single unit of deployment, or a single concern. Container should not be modified once it is started. No matter what reasons behind that - to deploy/undeploy applications, or modify/upgrade them. Instead, new image should be built with all required changes. After that stop existing container, and start a new one from created image.

Use tags appropriately.

Use immutable tags for production images. By immutable we understand specific tag that reference on same artifact during the time and this reference cannot be changed. Avoid

to use latest tag since it is not idempotent and you make builds are not idempotent**.** This means that when the result of each new build can be not the same as the previous one. The latest images today can be different from the latest image tomorrow or next month. It is critical to have reproducible deterministic behavior during rebuilding image.

Let's review some typical problems that developers or operations teams can face when using mutable tags. You are performing a **non-deterministic deployment** at any place where container is deployed using an image tag. There is no guarantee that the tag corresponds to the expected image. This applies to a simple *docker run*, or for orchestration tools like docker-compose, Kubernetes, etc.

Here is a scenario that can cause problem:

- Developer 1 executes a container on local machine using *image:**latest***. He pulls application version 1.2 and runs it.
- Several hours later, developer 2 pulls and runs the same *image:**latest*** on his local machine. He can be sure that this is the same version. Imagine situation that image maintainer published version 1.3 couple of minutes ago and *latest* tag is referencing to 1.3 now.
- Next mount, application was deployed to the production cluster. *myimage:**latest*** is now pointing to version 2.5. In this version some breaking changes was introduced. The application will crash in production.

Mutable tags are also useful in some scenarios:

- Mutable tag is used for artifact when it is not released in order to not create new version on each change
- *latest* (the default tag if not specified) is always pointing to the latest version of an image. *alpine* or *slim* is used to point to the latest corresponding versions.

- Quite common practice is using mutable tags to track versions in different environments (dev**,** prod**,** qa). Tags are updated when a new version is deployed to corresponding environment.

Some resources suggest to use digest for specifying docker image instead of named tag. It gives guarantee that images will always be the same. However, if you are using images from Docker hub from trusted vendors this recommendation might be redundant.

docker images --digests can be used to display the image digest. Example of such digest is *sha256:4f600a95fa1288f7b12198aa32ca00b4fb13b83b31533fa6b40499bd9bdf192f*).

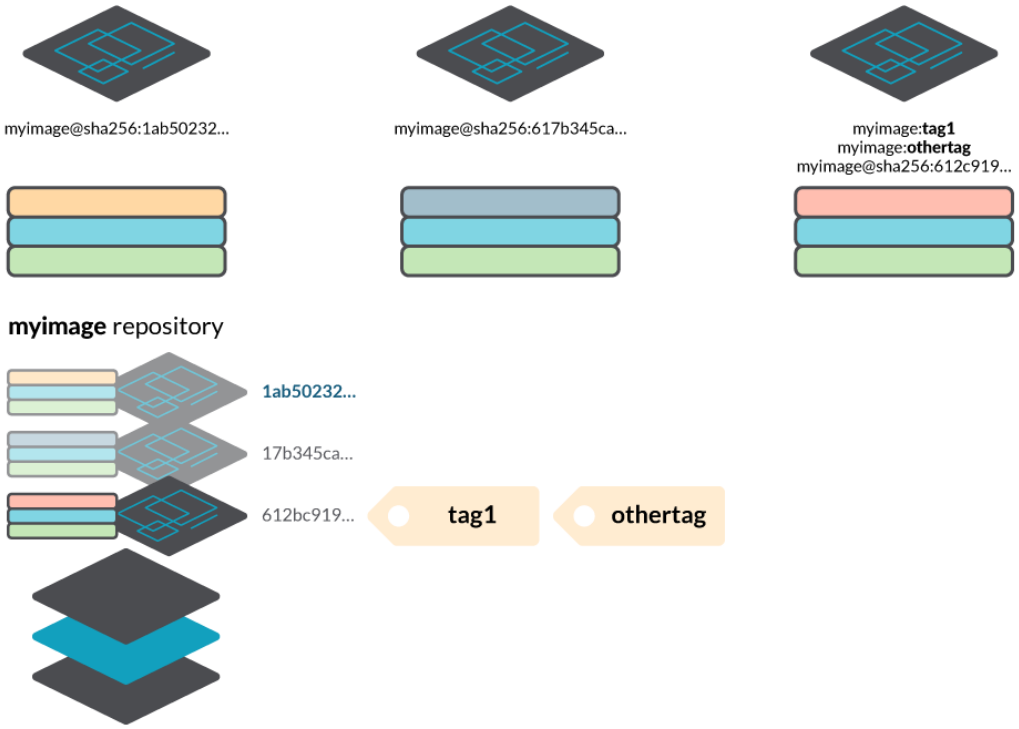

Figure 3.2.2. Docker tags explanation [18]

Configuration

Application should not depend on configuration – such as database usernames and passwords, or urls to external APIs. All this information should be provided at runtime. This is one of the most important things which is violated quite often. This means that configuration should be injected at runtime or other words provided by environment rather then hardcoded into code or into Docker image.

This separation concern forces you to have clear understanding what is configuration and what is code but at same time give you possibility to run same artifact or Docker image in multiple environments just provide appropriate configuration.

Let's consider some of the most common ways how configuration how configuration can be provided to a Java app in a container:

- Mount a volume containing configuration files. For example *.properties* file which can be red in Spring Boot using argument *--spring.config.additional-location*
- Set environment variables. Use *System.getEnv()* or application framework feature
- Use a network-based configuration service. For example, Spring Cloud Config [17]
- Provide arguments to the JVM on startup by overriding the container's *entrypoint.* Use *System.getProperty* or features provided by application framework

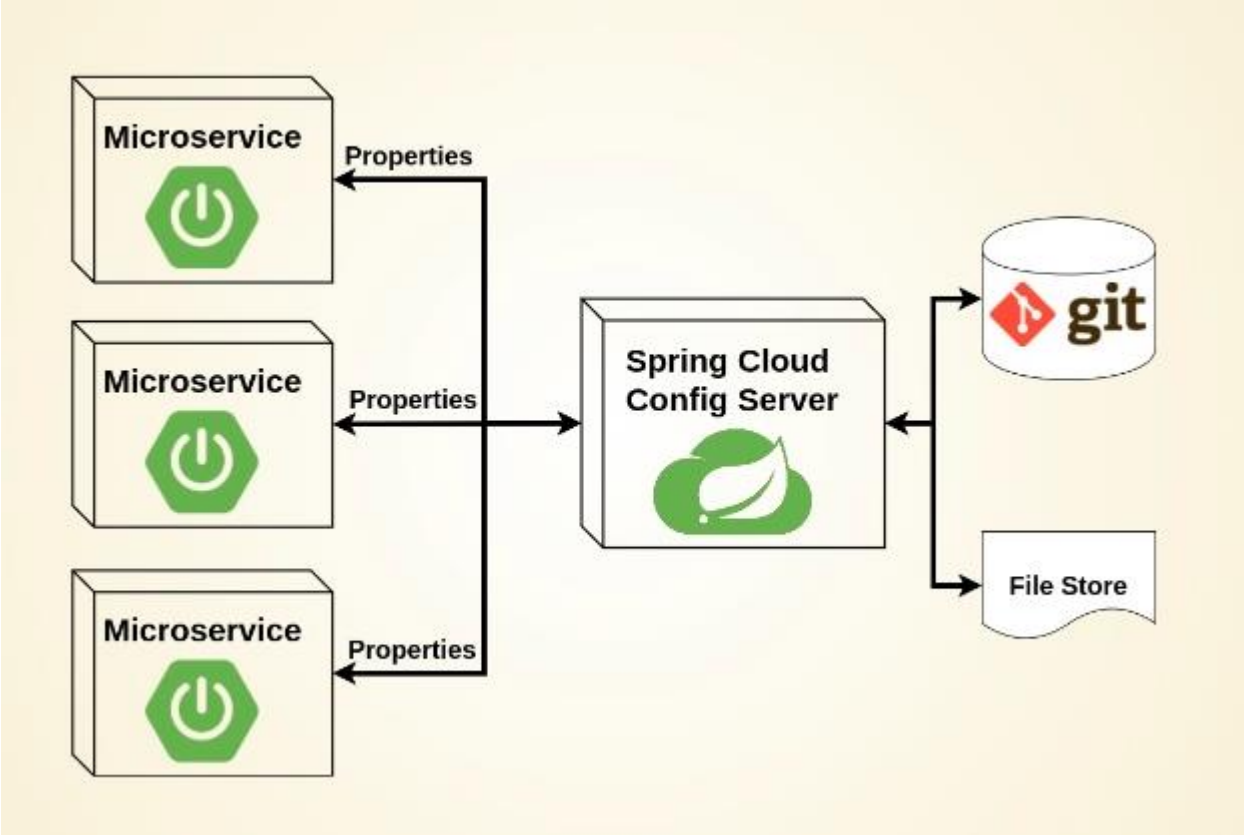

Figure 3.2.3. External configuration by Spring Cloud Config [19]

Following configuration externalization principle gives such benefit like portability. Having one image and providing any configuration as part of the environment to the container helps to be sure that same code is running on all environments. This makes debug issues or recreate environments much easier. But if we are going to build different Docker images for each target environment (dev, qa, prod) we will lose portability, because each container image is hardly tied to a specific environment.

Versioning

Developers must use versioning for images. It is recommended to use versioning for configurations files also.

Don't embed infrastructure into services.

All infrastructure staff like SSL termination, rate limiting, CORS and so on should be configured using infrastructure not application.

Quotas for services

Leaks in resources for single service should not crush machine on which other service might run. Also, it helps orchestrator to find appropriate node in cluster faster.

Java had some quirks while running in containers. It had some difficulties to understand how much memory it was able to consume. It often ended up consuming too much memory and it cases to container terminating by Docker. Sometimes it was unaware of the number of CPUs it was working with. These issues were resolved in more recent builds of Java. Thant is why so important to keep versions up to date. You need to make sure you’re using Java SE Development Kit 8, Update 191 (JDK 8u191), which was released in October 2018. It is recommended to use Java 11, which has significant memory and CPU improvements and is long-term supported release, or any other later version.

3.3. Security guidelines

Segregate services by security traits.

Services should be segregated by access type (public, internal and so on) and by required privileges. PoLP principle should be used (principle of least privileges).

The principle of least privilege states that any process, program, user should have only minimum privileges necessary to perform its function. For example, when we create user account to read or write data from database, we should not grand admin permissions to it. Developers should not have access to personal data records if they are writing line of codes in legacy project. The PoLP principle can also be referred to as the principle of minimal

privilege (PoMP) or the principle of least authority (PoLA). It is considered a best practice in information security to follow PoLP principle.

Let’s review the benefits of using PoLP principle:

- **Better security:** Edward Snowden was able to get millions of NSA files because his task for creating database backups had admin privileges. Because of the Snowden leaks, NSA to cut system administrators by 90 percent to limit data access [23].
- **Minimized attack surface:** Hackers got access to 70 million Target customer accounts. This is because of HVAC contractor who had permission to upload executable files. Since PoLP principle was not followed, broad attack surface was created.
- **Limited malware propagation:** Malware that infects a system which is designed in comply with principle of least privilege is often contained in small module only and is not propagated to other modules.
- **Better stability:** PoLP also increase system stability since it limits effects of changes to the module in which they’re made.
- **Improved audit readiness:** PoLP principle can reduce scope of audit. Interested point that, implementation of PoLP principle is required during audition by many common regulations.

Validate inbound data.

All incoming requests, responses, messages and events must be validated before application start to process them.

Don't expose sensitive data.

Attackers can use exposed sensitive data which may lead to fines from regulators. Showing stack trace can be used to enable negative impact on system.

Usually sensitive data is exposed as a result of not enough protecting data storage. It might be a result of weak encryption or even no encryption, software flaws or personal mistake when person uploads data to wrong server.

Control dependencies versions.

Dependencies must be updated regularly because updates nest fixes for vulnerabilities.

3.4. Maintainability guidelines

Use branching strategy.

Branching strategy helps developers work separately and do not affect each other. All changed in main branch should be done through pull requests.

Enable build automation.

Single build script for local developers' machine and remote automation builds must be used. All infrastructure tasks should be removed form build script. Example of infrastructure tasks: static code analysis, vulnerabilities analysis, etc.

Use tests appropriately.

A Testing Pyramid is an abstraction which has main idea to group software tests into different levels of detail. It gives a view of how many tests should be in each of these groups. Also, it helps developers and QAs create high-quality software and reduces the time required for developers to identify if a change they introduced breaks the code. Testing Pyramid helps to create more reliable test suite.

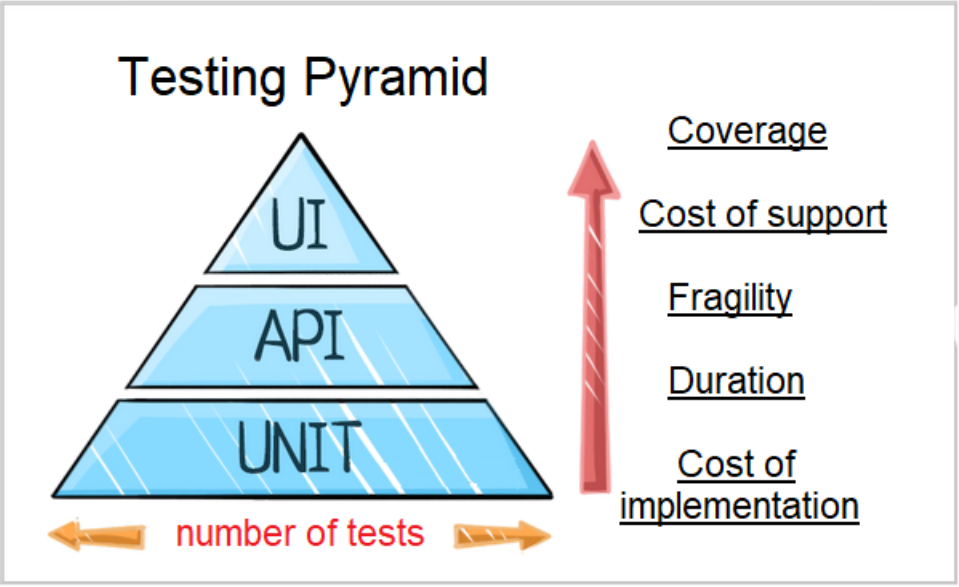

Figure 3.4.1. Classic Testing Pyramid [19]

There are 2 main principle that should be used:

- There should be tests of different details;
- The higher the level of pyramid, the fewer tests should be written

According to Testing Pyramid shape test suite should contains:

- lot of small and quick unit tests;
- more general tests;
- very few high-level end-to-end tests;

Let's review each level of Testing Pyramid.

Unit tests should form the major part of automation testing:

- Automation tasks are not closed until these scripts are launched on the implemented functionality;
- Developing along with unit tests forces developers to think about the problem they are solving and any edge cases they might face;

- Tests are detailed and can help pinpoint a defect;
- Runtime is fast because they don't need to rely on external systems like or UI;
- Unit tests are not expensive, easy to write, easy to maintain.

Integration tests should be in the middle of the pyramid. There are some aspect related to this level or tests:

- This layer is used to test business logic without using the user interface (UI);
- By testing outside of the user interface, you can test the inputs and outputs of APIs or services without all the complexity that the user interface introduces;
- These tests are slower and more complex than unit tests because they may need access to a database or other components.

UI tests are placed at the top of the pyramid. There are some aspects of UI tests:

- Most of your code and business logic should already be tested to this level;
- UI tests are written to ensure that the interface itself is working correctly;
- UI tests are slower and harder to write and maintain, so should be kept to a minimum.

However, classic testing pyramid can be extended to cover more different tests types and It can look like in a next way.

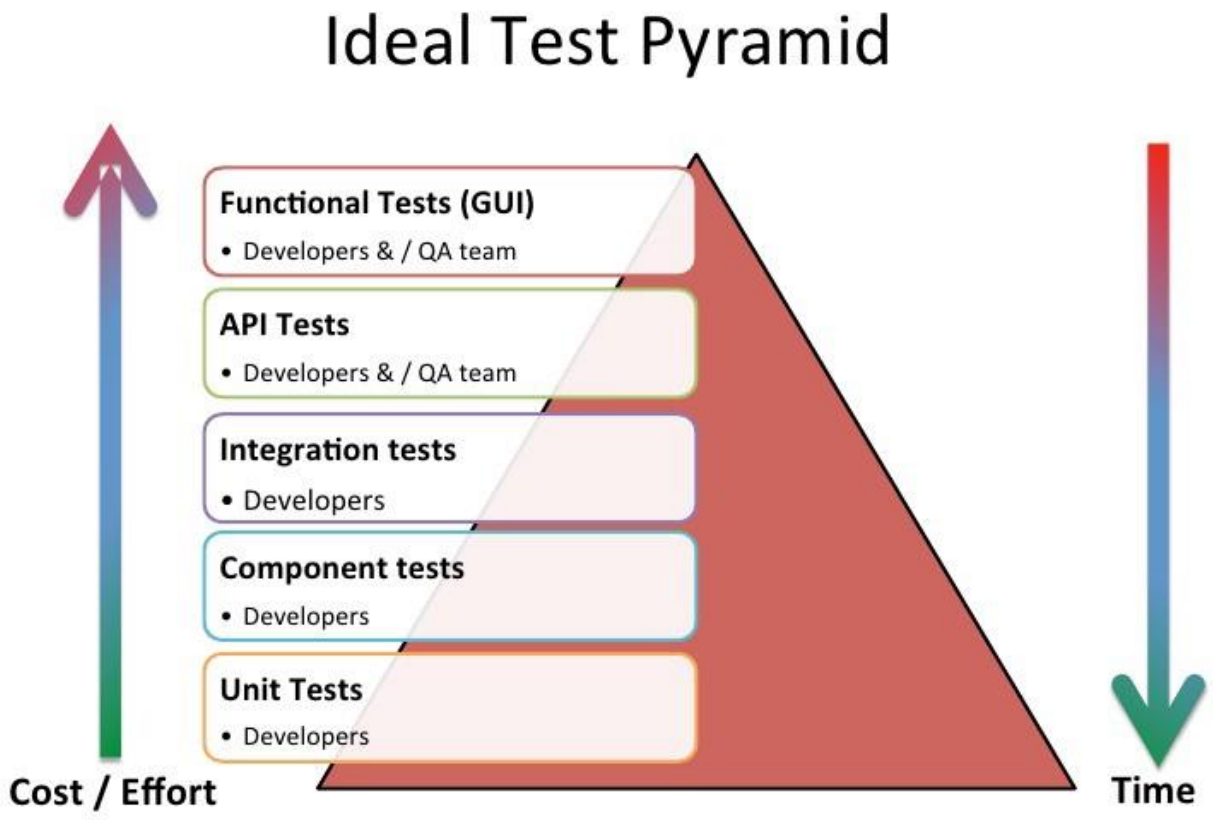

Figure 3.4.2. Ideal Testing Pyramid [40]

Define feedback activities.

Define all activities and quality gates that must be passed before code will be transferred to next stage. It helps shift feedback to the left part of feedback activity diagram and find errors earlier.

Use code conventions.

It improves readability of the code. It is recommended to have single convention in scope of all system but at least it should exist in scope of single team. Code convention template can be exported into file and shared with team. All modern IDEs support such feature.

Reduce code duplication.

It is recommended to try to have less than 3% of code duplication. Code review is required because static code analysis cannot find semantic code duplication that is.

Remove dead code.

It is not necessary to have unused or commented code because it can be reverted form git history.

Ensure methods and classes maintainability.

Use clean code principles. Robert C. Martyn in his book “Clean Code: A Handbook of Agile Software Craftsmanship” describe these principles.

3.5. Prototype

To demonstrate all the recommendations, we will consider simple example that consist of 2 java applications and 1 Kafka topic. It is an ordering system. First application is responsible for managing user orders. The second one is Warehouse service. When user want to create order, it provides all information about items and their amount to order service. Order service call Warehouse service in order to verify whether all items are available in required amount. After order was created Order service send corresponding event to Kafka topic. To start Kafka broker and Zookeeper we have *docker-compose-kafka.yml* in *order-service* repository in *docker* folder.

Let’s deep dive in prototype implementation. We have 2 java Spring Boot microservices. Order service calls REST endpoint if Warehouse service using Spring WebClient. When all necessary information for creating order is received it creates order and send event to Kafka broker.

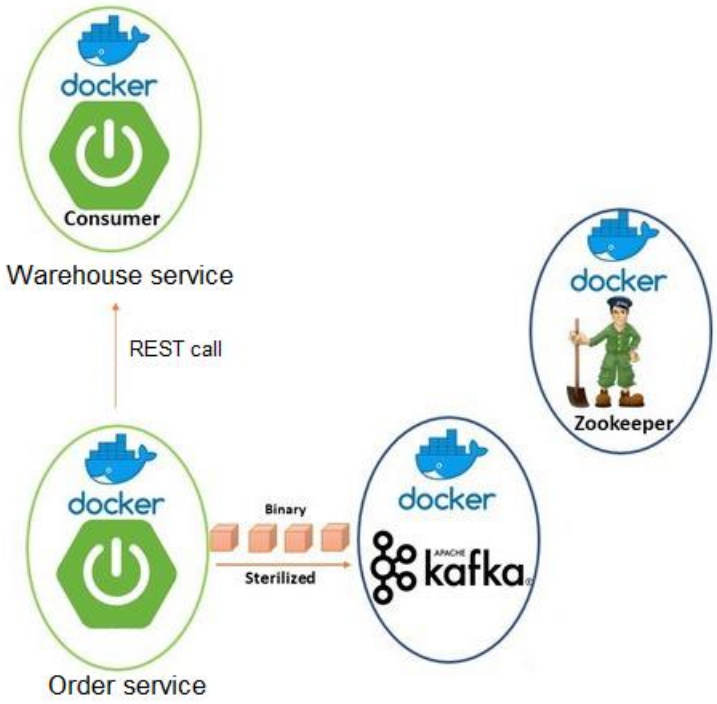

Figure 3.5.1. Project diagram

According to diagram Order service need to know URL of Warehouse service and Kafka broker. Since configuration must be externalized and not part of codebase, we will use environment variables in *application.yml* file.

```
server:
  port: 8090

order:
  service.warehouse.url: ${SERVICE_WAREHOUSE_URL:http://localhost:8091}

spring:
  kafka:
    producer:
      value-serializer: com.naukma.order.broker.KafkaSerializer
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS:localhost:9092}
    order-topic-name: ${ORDERS_TOPIC_NAME:orders_topic}
```

Let’s take a look what Docker file we will use for our project:

```
FROM openjdk:8-jre-alpine
COPY target/order-service-*.jar /deployment/order-service.jar
CMD ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar",
"/deployment/order-service.jar"]
```

And command how to run it:

```
docker build -t order-service -f docker/Dockerfile .
```

Build command should be executed in root project folder. There is no possibility to run command in Docker folder and specify relative path to jar file. From Docker documentation [15]: “The <src> path must be inside the context of the build; you **cannot ADD ../something**/something, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.” Dockerfile for Warehouse service the same except names.

Let’s build order and warehouse services

```
D:\Study\NaUKMA\Diplom\projects\order-service>docker build -t order-service:1.0.0 -f docker/Dockerfile .
[+] Building 0.2s (7/7) FINISHED
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 32B                                                                 0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 2B                                                                     0.0s
 => [internal] load metadata for docker.io/library/openjdk:8-jre-alpine                             0.0s
 => [internal] load build context                                                                   0.0s
 => => transferring context: 89B                                                                    0.0s
 => [1/2] FROM docker.io/library/openjdk:8-jre-alpine                                               0.0s
 => CACHED [2/2] COPY target/order-service-*.jar /deployment/order-service.jar                      0.0s
 => exporting to image                                                                              0.0s
 => => exporting layers                                                                             0.0s
 => => writing image sha256:ff861e65934aed7e36b8b78b55f9134974da31e8b0d24aaaf66f85875cffa6f4        0.0s
 => => naming to docker.io/library/order-service:1.0.0                                              0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

Figure 3.5.2. Docker image build of Order service

```
D:\Study\NaUKMA\Diplom\projects\warehouse-service>docker build -t warehouse-service:1.0.0 -f docker/Dockerfile .
[+] Building 0.3s (7/7) FINISHED
 => [internal] load build definition from Dockerfile                                                    0.0s
 => => transferring dockerfile: 243B                                                                    0.0s
 => [internal] load .dockerignore                                                                       0.1s
 => => transferring context: 2B                                                                         0.0s
 => [internal] load metadata for docker.io/library/openjdk:8-jre-alpine                                 0.0s
 => [internal] load build context                                                                       0.0s
 => => transferring context: 28B                                                                        0.0s
 => CACHED [1/2] FROM docker.io/library/openjdk:8-jre-alpine                                            0.0s
 => [2/2] COPY target/warehouse-service-*.jar /deployment/warehouse-service.jar                         0.1s
 => exporting to image                                                                                  0.1s
 => => exporting layers                                                                                 0.0s
 => => writing image sha256:712c7d3454ab1984f096b3b8ec1b19e91598b5ced908058a5844733d53e5d61c            0.0s
 => => naming to docker.io/library/warehouse-service:1.0.0                                              0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

Figure 3.5.3. Docker image build of Warehouse service

Now we are going to start whole platform. For this purpose, *docker-compose.yml* file was created.

```
version: '2'
services:
 zookeeper:
  image: wurstmeister/zookeeper:3.4.6
  expose:
   - "2181"

 kafka:
  image: wurstmeister/kafka:2.11-2.0.0
  depends_on:
   - zookeeper
  ports:
   - "9092:9092"
  environment:
   KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
   KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
   KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181

 warehouse-service:
  image: warehouse-service:1.0.0
  ports:
   - "8091:8091"

 order-service:
  image: order-service:1.0.0
  depends_on:
   - warehouse-service
   - kafka
  ports:
   - "8090:8090"
```

```
environment:
  ORDERS_TOPIC_NAME: orders_topic
  SERVICE_WAREHOUSE_URL: http://warehouse-service:8091
  KAFKA_BOOTSTRAP_SERVERS: kafka:9092
```

To configure Order service environment variables should be set. The same image can be used for different environments (dev, uat, prod) only environment variables must be changed. Using *docker-compose up* command all service can be started.

In Order service communicate with Warehouse service via REST is implemented so we will use HTTP header *X-Correlation-Id* to send value. Also, we need to analyze whether we already receive correlation ID and if not generate new one. For this purpose, we will use *CorrelationHeaderFilter*, which is a standard Java EE Filter that inspects the *HttpServletRequest* header for the presence of a correlation id. *ThreadLocal* variable will be used to store value and we will write wrapper class *RequestCorrelation* to work with it. If a correlation id is not found then *CorrelationHeaderFilter* generate new one. It is necessary to make a note that this code will work on synchronous communication only. To work with correlation id when you handle requests in non-blocking way *DeferredResult* class in combination with the Servlet 3 asynchronous support can be used but we have to deal with storing correlation id since thread that initially handles the request will not be the thread doing the actual processing and *ThreadLocal* variable approach will not work. This is separate topic and will not be covered in this work.

Also, we have to propagate correlation id to message that we send to Kafka topic. Wo this purpose we create so called envelop message *KafkaMessageEnvolope* class which consist of *KafkaMessageInfo* and data itself. Data is child class of *KafkaSerializable*. Each time when we are going to send new event, we wrap data in *KafkaMessageEnvolope* and add all context information. In our case it is correlation id but other context staff can be added if needed.

Let’s try to call Order service in order to see how correlation id is propagated.

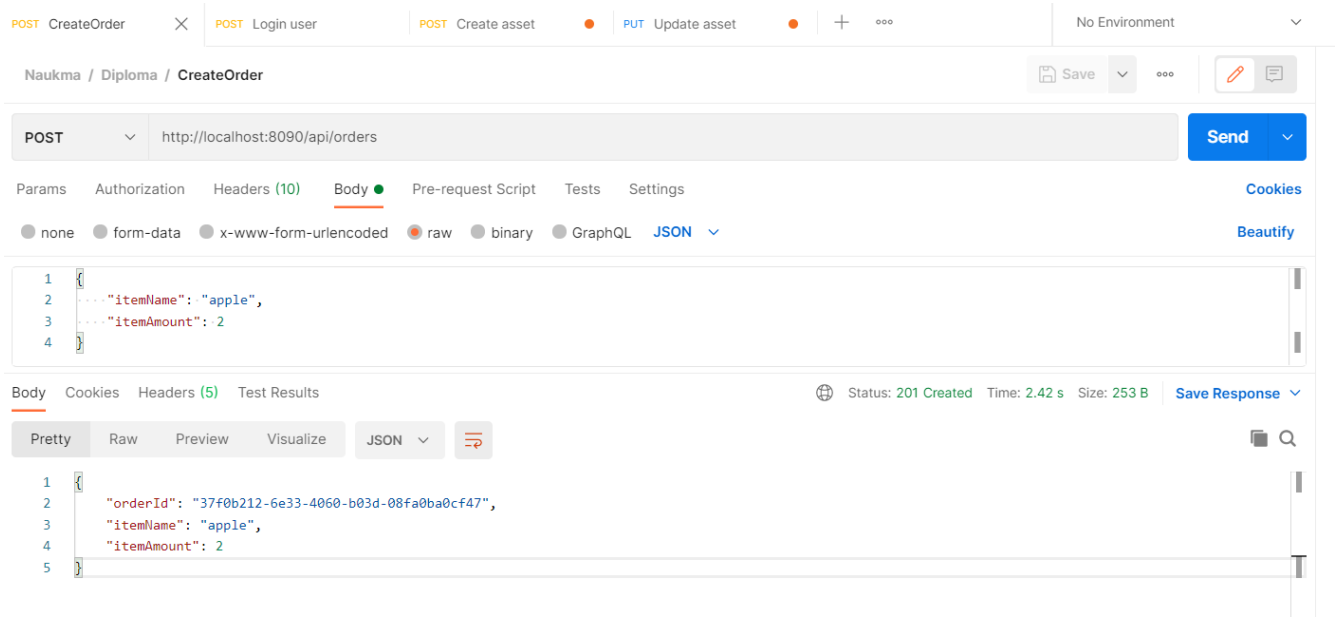

Figure 3.5.4. Consuming Kafka messages

To read message from kafka topic we will use consumer from kafka bin folder. Using docker command we need to get access to kafka container bash.

After that we will print services logs into *stdout*. First of all, we need to know containers ids. We can do this using *docker p*s command.

```
D:\Study\NaUKMA\Diplom\projects\order-service>docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS                                NAMES
e9ccfba82e58   order-service:1.0.0            "java -Djava.securit…"   4 minutes ago   Up 4 minutes   0.0.0.0:8090->8090/tcp               docker_order-service_1
1ef8aaa91608   wurstmeister/kafka:2.11-2.0.0  "start-kafka.sh"         4 minutes ago   Up 4 minutes   0.0.0.0:9092->9092/tcp               docker_kafka_1
4e3726a9b76a   wurstmeister/zookeeper:3.4.6   "/bin/sh -c '/usr/sb…"   4 minutes ago   Up 4 minutes   22/tcp, 2181/tcp, 2888/tcp, 3888/tcp docker_zookeeper_1
45f88ff7c4a5   warehouse-service:1.0.0        "java -Djava.securit…"   4 minutes ago   Up 4 minutes   0.0.0.0:8091->8091/tcp               docker_warehouse-service_1
```

Figure 3.5.5. List of running containers

docker logs 45f88ff7c4a5

```
2021-05-28 16:06:13.698  INFO 1 --- [nio-8091-exec-1] c.n.warehouse.controller.ItemController  : Received request with
 correlationId: ae6b7805-e6ed-48ea-8374-4e4e47141fdd
```

Figure 3.5.6 Order service logs

docker logs 45f88ff7c4a5

```
2021-05-28 16:06:13.698  INFO 1 --- [nio-8091-exec-1] c.n.warehouse.controller.ItemController  : Received request with
 correlationId: ae6b7805-e6ed-48ea-8374-4e4e47141fdd
```

Figure 3.5.7 Warehouse service logs

To read kafka messages we will use kafka-console-consumer utility.

docker exec -it 1ef8aaa91608 /opt/kafka/bin/kafka-console-consumer.sh --topic orders_topic --from-beginning --bootstrap-server localhost:9092

```
D:\Study\NaUKMA\Diplom\projects\warehouse-service>docker exec -it 1ef8aaa91608 /opt/kafka/bin/kafka-console-consu
mer.sh --topic orders_topic --from-beginning --bootstrap-server localhost:9092
{"messageInfo":{"correlationId":"ae6b7805-e6ed-48ea-8374-4e4e47141fdd"},"data":{"type":"OrderCreatedEvent","order
Id":"37f0b212-6e33-4060-b03d-08fa0ba0cf47","itemName":"apple","itemAmount":2}}
```

Figure 3.5.8. Consuming Kafka messages

As we can see correlation id was generated on Order service side and propagated to all other components of the system.

For correlation id logic 2 java classes are responsible. One of them is *RequestCorrelation* which is correlation id value holder.

```
public class RequestCorrelation {

   public static final String CORRELATION_ID_HEADER = "X-Correlation-Id";

   private static final ThreadLocal<String> id = new ThreadLocal<String>();

   public static String getId() {
      return id.get();
   }

   public static void setId(String correlationId) {
      id.set(correlationId);
   }
}
```

The second one is filter which is listening incoming requests trying to retrieve value from *X-Correlation-Id* header. If such header is missing, new value will be generated. After that value is stored in *ThreadLocal* variable in *RequestCorrelation* class.

```
@Component
public class CorrelationHeaderFilter implements Filter {

  private static final Logger log = LogManager.getLogger(CorrelationHeaderFilter.class);

  @Override
  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
      throws IOException, ServletException {

    final HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
    String currentCorrId = httpServletRequest.getHeader(RequestCorrelation.CORRELATION_ID_HEADER);

    if (!currentRequestIsAsyncDispatcher(httpServletRequest)) {
      if (currentCorrId == null) {
        currentCorrId = UUID.randomUUID().toString();
        log.debug("No correlationId found in Header. Generated : " + currentCorrId);
      } else {
        log.debug("Found correlationId in Header : " + currentCorrId);
      }

      RequestCorrelation.setId(currentCorrId);
    }

    filterChain.doFilter(httpServletRequest, servletResponse);
  }

  private boolean currentRequestIsAsyncDispatcher(HttpServletRequest httpServletRequest) {
    return httpServletRequest.getDispatcherType().equals(DispatcherType.ASYNC);
  }
}
```

Common logic to work with correlation id moved to correlator library since it is used in both Order and Warehouse services. This is Spring Boot starter and to start using this logic we need to just add dependency in pom file.

```
<dependency>
  <groupId>com.naukma.common</groupId>
  <artifactId>correlator</artifactId>
  <version>1.0.0</version>
</dependency>
```

Let’s try to send request which will fail during validation in order to see error response.

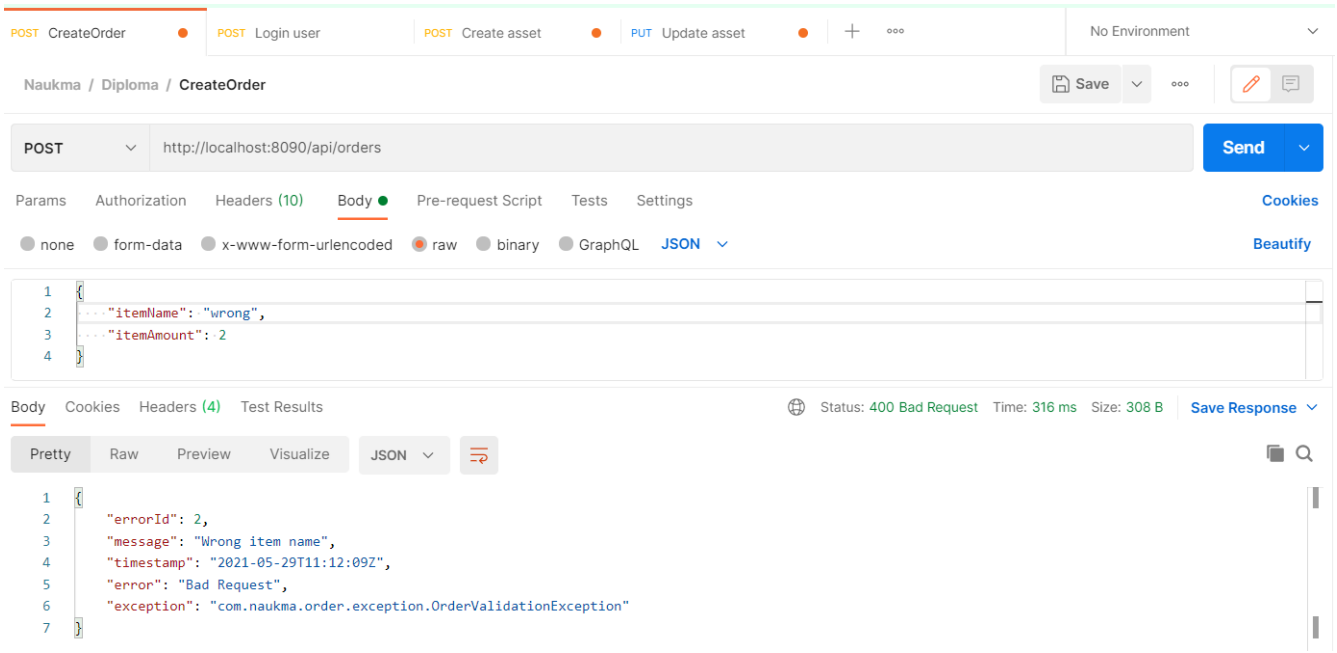

Figure 3.5.9. Order service error message

In our project we defined next error message format:

- *erorrId* – business error code
- *message* – meaningful human-readable message
- *timestampt* – data and time in ISO time format when error happens
- *exception* – since we use Java for building our service in this field, we have class name of exception which occurs

There are several approaches how to implement Exception Handling with Spring Boot for a REST API. We will use *@ControllerAdvice* annotation. We have *ErrorHandlingController* class where we address all exceptions that happen during service execution. Each exception is handled by separate method. Here we can see how *OrderValidationException* is processed:

Each exception extends *AbstractOrderServiceException* with *OrderPlatformError* which is enumeration error code representation.

```
public abstract class AbstractOrderServiceException extends RuntimeException {

    private final OrderPlatformError error;

    public AbstractOrderServiceException(OrderPlatformError error, String message) {
        super(message);
        this.error = error;
    }

    public OrderPlatformError getError() {
        return error;
    }
}

public enum OrderPlatformError {

    ORDER_NOT_FOUND(1),
    WRONG_ORDER_DATA(2),

    INTERNAL_SERVER_ERROR(500);

    private final int id;

    OrderPlatformError(int id) {
        this.id = id;
    }

    public static OrderPlatformError getErrorMessageById(int id) {
        for (OrderPlatformError orderPlatformErrorMessages : values()) {
            if (orderPlatformErrorMessages.getId() == id) {
                return orderPlatformErrorMessages;
            }
        }
        return null;
    }

    public int getId() {
        return id;
    }

}
```

For handling all errors *ErrorHandlingController* was implemented. It was annotated *@ControllerAdivce* annotation. Each error handled by corresponding method.

```
@ControllerAdvice
public class ErrorHandlingController {

   private static final String TIMESTAMP_FIELD_NAME = "timestamp";
   private static final String EXCEPTION_FIELD_NAME = "exception";
   private static final String ERROR_FIELD_NAME = "error";
   private static final String ERROR_ID_FIELD_NAME = "errorId";
   private static final String MESSAGE_FIELD_NAME = "message";

   private static final Logger log = LogManager.getLogger(ErrorHandlingController.class);

   @ExceptionHandler(ObjectNotFoundException.class)
   @ResponseBody
   public Map handleObjectNotFoundException(ObjectNotFoundException exc, HttpServletResponse response) {
      return buildError(exc, response, HttpStatus.NOT_FOUND);
   }

   @ExceptionHandler(OrderValidationException.class)
   @ResponseBody
   public Map handleChatValidationException(OrderValidationException exc, HttpServletResponse response) {

      return buildError(exc, response, HttpStatus.BAD_REQUEST);
   }

   @ExceptionHandler(MethodArgumentNotValidException.class)
   @ResponseBody
   public Map handleJsonSchemaValidationExceptions(MethodArgumentNotValidException exc,
HttpServletResponse response) {
      ImmutableMap.Builder mapBuilder =
            getMapBuilderWithErrorMessageIdPair(OrderPlatformError.INTERNAL_SERVER_ERROR, exc.getMessage());
      return buildError(exc, response, HttpStatus.BAD_REQUEST, mapBuilder);
   }

   @ExceptionHandler(Exception.class)
   @ResponseBody
   public Map handleException(Exception exc, HttpServletResponse response) {
      ImmutableMap.Builder mapBuilder =
            getMapBuilderWithErrorMessageIdPair(OrderPlatformError.INTERNAL_SERVER_ERROR, exc.getMessage());
      return buildError(exc, response, HttpStatus.INTERNAL_SERVER_ERROR, mapBuilder);
   }

   private Map buildError(AbstractOrderServiceException exc, HttpServletResponse response, HttpStatus httpStatus)
{
      ImmutableMap.Builder mapBuilder =
            getMapBuilderWithErrorMessageIdPair(exc.getError(), exc.getMessage());
      return buildError(exc, response, httpStatus, mapBuilder);
   }

   private Map buildError(Exception exc,
                          HttpServletResponse response,
                          HttpStatus httpStatus,
                          ImmutableMap.Builder<String, Object> mapBuilder) {
      log.error(exc.getMessage(), exc);
      response.setStatus(httpStatus.value());

      mapBuilder
            .put(TIMESTAMP_FIELD_NAME, DateUtils.nowAsIsoTimeFormatString())
            .put(ERROR_FIELD_NAME, httpStatus.getReasonPhrase())
```

```
            .put(EXCEPTION_FIELD_NAME, exc.getClass().getCanonicalName());

        return mapBuilder.build();
    }

    private ImmutableMap.Builder<String, Object> getMapBuilderWithErrorMessageIdPair(OrderPlatformError error,
String message) {
        ImmutableMap.Builder<String, Object> mapBuilder = ImmutableMap.builder();
        return mapBuilder
            .put(ERROR_ID_FIELD_NAME, error.getId())
            .put(MESSAGE_FIELD_NAME, message);
    }
}
```

Since error handling logic used in both service and new Java services will need this functionality also, new Spring Boot started was created. Just adding dependency is enough to get all functionality.

```
<dependency>
  <groupId>com.naukma.common</groupId>
  <artifactId>error-handler</artifactId>
  <version>1.0.0</version>
</dependency>
```

In our project we are going to use default Spring Actuator implementation. This would be enough for our purposes. To start using Spring Actuator we just need to add dependency to maven pom file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Let’s see what Spring Boot Actuator provides in default configuration. We need to call GET /actuator endpoint.

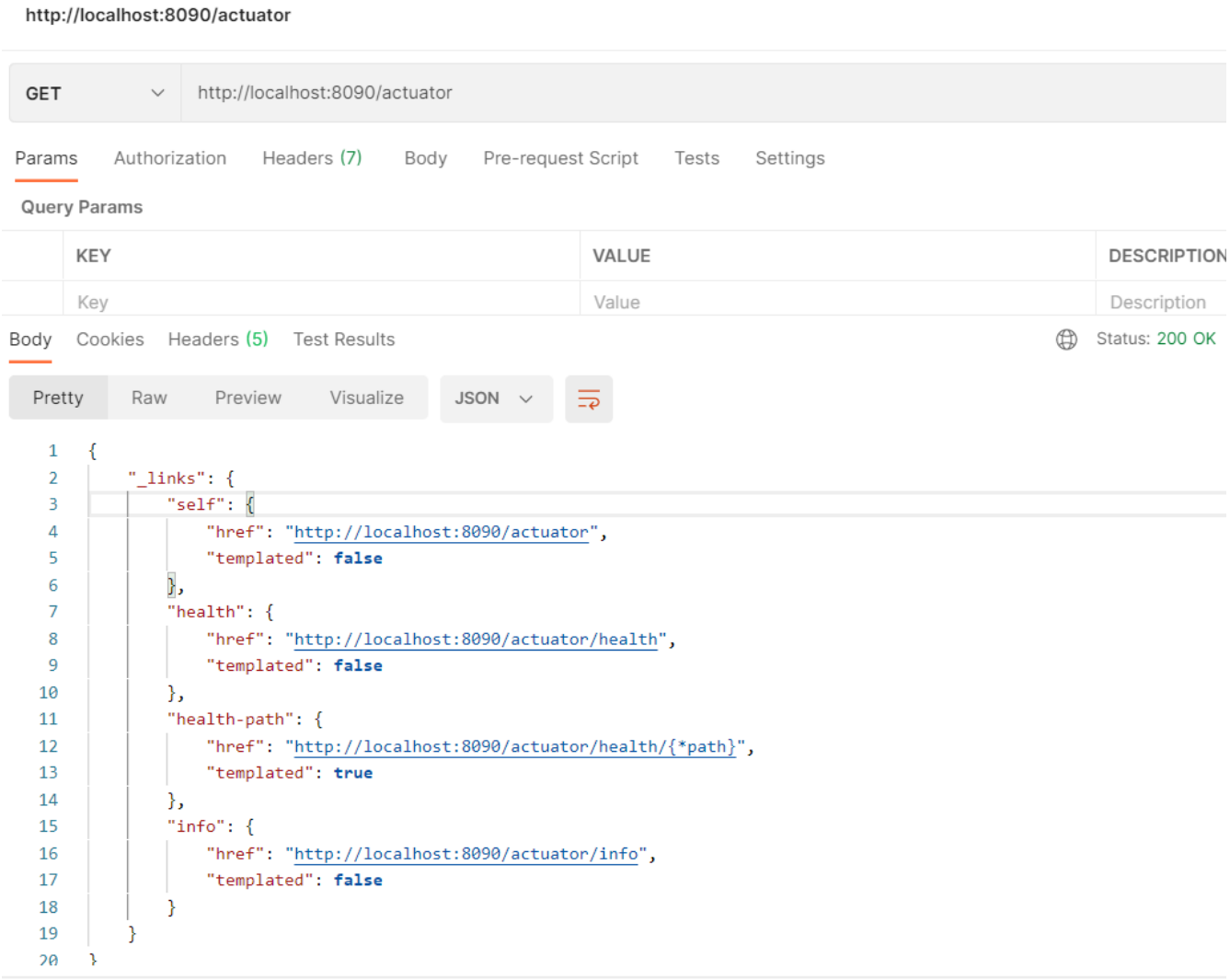

Figure 3.5.10. Actuator endpoint response

We are interested in /actuator/health endpoint. We do not have any orchestration tool but on real environment we most probably have some like Kubernetes, ECS or similar one which will be using this endpoint to understand the status of service.

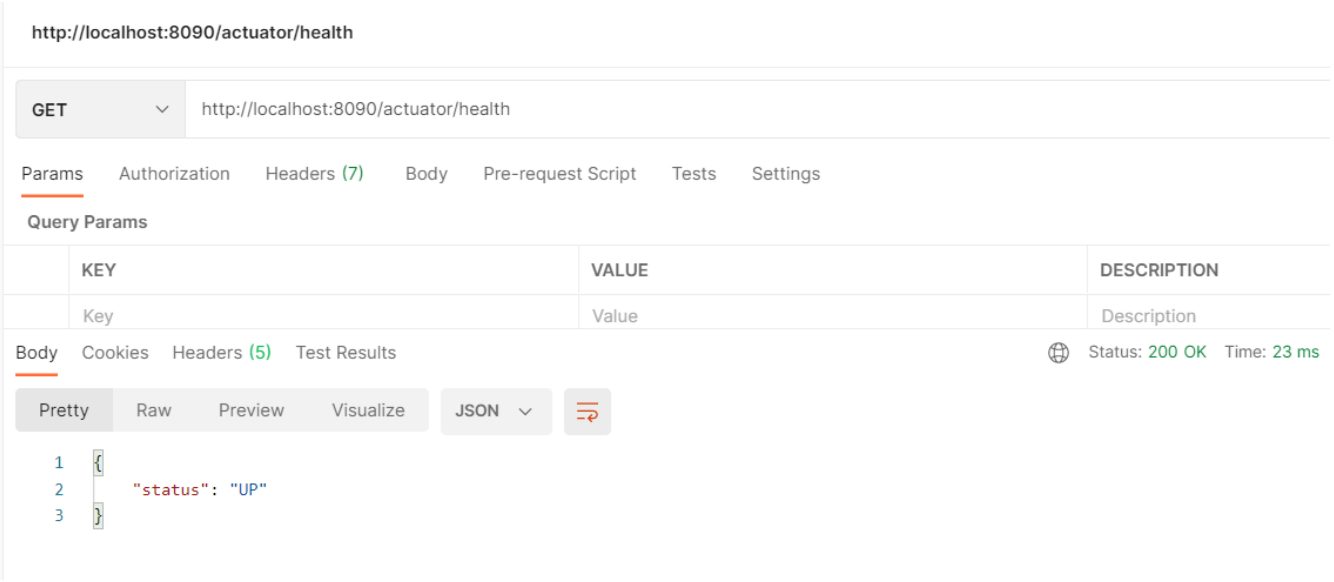

Figure 3.5.11. Health endpoint response

Results

In first part of this paper we defined what is microservice architecture, gave basic aspects of such type of architecture and differences between monolithic architecture. Reviewed existing twelve-factor application methodology with explanation of each principle.

The second part is dedicated to requirements to services written using microservices architecture. Questions what is the definition of requirements, what is the classification of requirements with explanation of each one. Also, this part contains material about quality attributes with definition of some of the most important one.

The third part contains guidelines grouped by quality attributes with detailed explanation and example of realization for synthetic project written in Java language. Real case example of issues that can happen if some important principle is ignored are provided.

Existing methodology using guidelines can be used for building products using microservices architecture. These guidelines can be extended further to cover aspects more deeply.

References

1. https://www.oreilly.com/library/view/building-maintainable-software/9781491955987/ch01.html
2. https://www.researchgate.net/publication/245390014_Reliability_and_maintainability_allocation_to_minimize_total_cost_of_ownership_in_a_series-parallel_system
3. https://medium.com/@flexbasenet/topic-software-maintainability-checklist-for-software-architects-44527ae5f2af
4. https://martinfowler.com/articles/microservices.html
5. https://businessanalyst.techcanvass.com/types-of-requirements-as-per-babok/
6. IIBA, *A Guide to the Business Analysis Body of Knowledge (BABOK Guide) 3rd Edition*
7. Oleksii Zhylenko, *Distributed Systems Technical Audit*
8. https://12factor.net/
9. https://www.magalix.com/blog/cloud-native-logging-and-monitoring-pattern
10. https://datatracker.ietf.org/doc/html/rfc7807
11. https://developers.facebook.com/docs/graph-api/using-graph-api/error-handling
12. https://thenewstack.io/monitoring-and-observability-whats-the-difference-and-why-does-it-matter/
13. http://openjdk.java.net/projects/portola/
14. https://continuousdelivery.com/
15. https://docs.docker.com/engine/reference/builder/
16. https://openliberty.io/blog/2018/06/29/optimizing-spring-boot-apps-for-docker.html
17. https://spring.io/projects/spring-cloud-config
18. https://sysdig.com/blog/toctou-tag-mutability/
19. https://medium.com/walmartglobaltech/manage-application-configuration-using-spring-cloud-config-80b27ecb34b7
20. Steve McConnell: *Code Complete*
21. https://dzone.com/articles/an-overview-of-health-check-patterns

22. https://habr.com/ru/post/258739/
23. https://www.reuters.com/article/us-usa-security-nsa-leaks-idUSBRE97801020130809
24. https://medium.com/@a0x8o/real-time-performance-profiling-analytics-for-microservices-using-apache-spark-96d026083021
25. https://www.bmc.com/blogs/observability-vs-monitoring/
26. https://owasp.org/www-project-top-ten/
27. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
28. Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy, *Site Reliability Engineering: How Google Runs Production Systems*
29. https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-tag-mutability.html
30. https://google.github.io/styleguide/javaguide.html
31. Robert C. Martyn, *Clean Code: A Handbook of Agile Software Craftsmanship*
32. https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf
33. https://jeffkreeftmeijer.com/git-flow/
34. https://medium.com/responsetap-engineering/monolith-to-microservices-to-serverless-our-journey-745a2b9620ec
35. https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
36. https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9
37. https://dev.to/jignesh_simform/evolution-of-serverless-monolithic-microservices-faas-3hdp
38. https://deepsource.io/blog/exponential-cost-of-fixing-bugs/

39. https://dzone.com/articles/go-microservices-part-12-distributed-tracing-with

40.https://qastart.by/mainterms/64-pirama-testov-testirovaniya