CORE INTEROPERABILITY STANDARDS

WHAT IS REST? #LORENZO

REST (an acronym for REpresentational State Transfer) was proposed as an architectural style which featured the network of web pages as a “distributed” finite state machine, where actions result in a transition to next state. It also featured the concept of “state representation”. REST is, in practice, a redesign of the HTTP protocol in order to feature:

1. **Resource identification through URIs:** The server exposes a set of resources, which are identified by predefined URIs.
2. **Uniform Interfaces:** The possible operations are just the HTTP ones (so PUT, GET, POST, DELETE, where put creates a resource and post modifies the state of one).
3. **Self descriptive messages:** As the approach is resource-centric, resources are decoupled from their representation so that their content can be accessed in a variety of formats (JSON, XML…).
4. **Stateful interactions through URIs:** HTTP is a stateless protocol, but a stateful interaction can be achieved via responding to requests (on the server side), with a new state.

HOW CAN WE CREATE/UPDATE/ACCESS RESOURCES IN REST? #LORENZO, #GDL, #LEO

The methods that allow creation, updation and access are the standard HTTP methods. In particular the http method:

- GET is used to retrieve a resource
- PUT is used to update/create* one (idempotent)
- DELETE is used to delete one
- POST is used to create/update* one (not idempotent)

This is more “pragmatics” than an actual rule, in general, every REST API should offer some sort of documentation in order to understand what is the access pattern. Another note should be that in order to interact with a resource the client needs to know the URI (at least one, to which might follow a stateful interaction based on “redirecting”).

* *http://restcookbook.com/HTTP%20Methods/put-vs-post/*

WHICH ARE THE PROS AND CONS OF REST? #LORENZO

REST allows for a simpler (more human intuitive), more scalable and more efficient communications with respect to standard WS standards like SOAP. The learning curve is very low since there are a lot of experimented tools that allow to turn existing application into RESTful services by providing and exposing HTTP APIs. The exploit of a stateless protocol like HTTP allows for a great scalability degree. This also allows for an easier testing/deployment phase, (by using a Web Browser or tools like Postman).

On the opposite side, REST brings also some disadvantages. Complex interfaces are hard to manipulate via URLs, both because of their fixed length, and because the resource tree to be designed might become very complex. Also there is no “best practise” manual to design good REST APIs, and also no semantic rules are provided in order to fix the functionalities of HTTP methods, so potentially one service could make a resource deletable/modifiable/creatable via a GET method.

QoS management and monitoring must be implemented “by hand”, in opposition to WS-*. In addition to that, the design of the resource specification and the URI addressing scheme might become challenging in REST. In conclusion REST performs well for human application (humans read manuals), while for machine readable and understandable application, together with enterprise ones, WS-* still performs better.

WHAT IS SOAP? #LORENZO

SOAP (Simple Object Access Protocol) is an XML based standard, built over HTTP that allows communication between heterogeneous systems. The primary use case would be inter-application communications. A SOAP message is formed by an envelope, which wraps an optional header and a mandatory body. An header can have any number of header entry, each specifying meta information regarding message processing and the role of the receiving entity. The typical process (which has to be described in the SOAP header), is that a SOAP message is sent by a sender, processed by any number of SOAP intermediary (process nodes), and gets received by a SOAP receiver. This allows for a role based approach to message handling (where each node is identified by a URI), where the presence of the optional attribute “mustUnderstand”, specifies if the processing node has to mandatorily respect the specifications of the SOAP header when processing (therefore understanding its content), or either send a fault message. The SOAP body instead, contains the payload of the message, which may either be a fault, or a normal payload. In case it is not a fault, then this might be a document type (pure XML payload) or RPC type (Remote Procedure Call).

HOW ARE SOAP MESSAGES TRANSPORTED? #LORENZO

SOAP messages are transported in HTTP requests and response, and have HTTP URLs as endpoints. The HTTP payload is XML based. Each SOAP message is formed by an envelope, which wraps an optional header and a mandatory body. The header contains informations regarding the processing of the message. It might contain informations like where the document must be sent, digital signatures, and allows for additional SOAP intermediaries (intermediate SOAP processing nodes). The SOAP body contains either some payload or a fault message. It allows for two communication styles:

1. Remote Procedure Call (RPC). An RPC web service appears as a remote object, clients send requests via method calls, and receives a response from the web service (might receive a fault).
2. Document style: Message oriented, it is just XML payload.

It supports for two types of encoding: Literal and encoded (although Document style encoding is not used in practice). Encoding is useful in order to have a consistent serialization schema for RPC kind of access to objects.

WHAT IS WSDL? #LORENZO

WSDL, Web Service Description Language, is a machine readable XML based language which allows the description of web services. It is used to support SOAP (which only allows the formatting and the processing of messages, but doesn't have any way to know the operations that are exposed by a service), and is useful to define the interfaces of web service, together with the transport protocol that given service uses. The three main points of focus for WSDL are:

1. description of the operations provided by a service
2. the URI where the service resides
3. description of the way to invoke those functions (transport protocol and parameters)

It is organised in interfaces (similar to the programming language ones), which is formed by an abstract service description (with operations, parameters, abstract data types…), and a concrete endpoint implementation (binds the abstract interface to the network address, the protocol and specific data structures).

WHAT IS A REQUEST-RESPONSE OPERATION IN WSDL? #LORENZO

Request-response is one kind of two-way message exchange possibile in WSDL and is the typical case of remote procedure call. The sender will send a SOAP request to the service, which will send back a response. In the WSDL interface the operation is declared with an input element, and is followed by an output element. The client here (by client I mean the one entity initiating the communication) is NOT the web service, but is the procedure caller (in opposition with solicit-response message exchange pattern).

WHAT IS A PORTTYPE / PORT IN WSDL? #LORENZO

A Port Type in WSDL is a part of the abstract interface, and it defines the service interface, as an abstract set of operations, by providing the name, and the XML tags input/output.
The port, instead, is part of the concrete interface (and of the Service section in particular). It associates a binding (a concrete protocol and a data format specification for each operation defined in the port-type section) to an actual network address. It contains information regarding physical address and protocol used.

MICROSERVICES

WHICH ARE THE MAIN CHARACTERISTICS OF MICROSERVICES-BASED ARCHITECTURES? #LORENZO

The main feature of microservice based architectures are:

- **Componentization via Services:** The application gets decomposed in services, which are different from libraries because they are out of process-components. This is ideal because each service is independently deployable (with respect to the case of libraries, for which this property doesn't hold), and this approach forces a clear per-component interface.
- **Organized around business capabilities:** Microservices allow for a cross-functional team based organization, which means every team has a full range of competences (UI specialists, DB administrators, middleware specialists…).
- **Products, not projects:** This, together with the cross functional stack, is a typical Dev-Ops approach, in which the Amazon notion "You build it, you run it" is the mantra. So the developer teams is not focused on a project to end, and to deliver to the operators, but rather on a product, and the team has also the duty to interact with the customers.
- **Smart endpoints and dumb pipes:** The logic of microservices resides in microservices themselves. The communication pipes are usually simple "dumb" protocols like REST, or asynchronous queues like Rabbit-MQ, in opposition with SOAP's approach, where communication channels (ESP alias Enterprise Service Bus) are even able to assure QoS requirements.
- **Decentralized governance:** Microservices allow for a "polyglot" environment, where different programming languages and different technologies can be adopted, and not clash with each others. This is made possible by the "dumb pipe" mechanism in between services. This approach drives also a different mentality, which is the typical one of microservice teams, that are usually reluctant to big standards, and prefer providing useful tools (like Netflix's chaos monkey), so that other microservice teams can use them when facing similar problems.
- **Decentralized Data Management:** Every microservice gets its own database (and is responsible of handling it). This approach brings also the problem of having to handle a distributed state. This is usually handled with the concept of eventual consistency (with respect to distributed transactions). This is the choice typically because of shorter response time, always considering the fact that the trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency.
- **Infrastructure Automation:** This property is related to the concepts of CI (continuous integration) and CD (continuous delivery), where build-deployment-management phases of a microservice are automated by a set of Dev-Ops available tools.
- **Design for Failure:** The "Services will fail" mantra is usually adopted in microservice based environments, because in a wide and distributed environment, where each component has a given QoS threshold, failure is not an exception, it is the norm. The idea is to design services which are "failure" aware, and are even stressed by tools that automatically inject failures on services (like for netflix simian army). So there is emphasis on the monitoring phase of a microservice lifespan, in order to detect as soon as possible a failure, in order to warn the development team. Patterns that provide elegant solutions to failures, like for example the circuit breaker, are usually adopted.
- **Evolutionary design:** Microservices can also be "plugged" in a monolithic contex, like it might be a case for a news website. In case of a particular event, a new thematic section of the site can be added, and this can be handled by a microservice without touching the main application code. This approach leads to an evolution aware design, where in case of a requirement evolution, just a single (eventually new) component needs to be deployed in order to add a new feature.

WHICH ARE THE MAIN PROS AND CONS OF MICROSERVICES? #LORENZO

PROS:

- **Shorter lead time:** There are a lot of tools available to automate a lot of lead phases. In addition to that, every service is independently deployable, so there is no need to re-deploy the whole application for changes on a sole microservice. Lead time is also shorter because of the reduction of cross-team requests (chords between functional silos), which is a bottleneck in classic software engineering. By having autonomous full-stack team, those dependencies are reduced hugely in number.
- **Effective Scaling:** Microservices allow for horizontal scalability, which means that it is possible to scale up or down only the services which are the bottleneck of the application at any given time.
- **Decentralized data management:** Every team gets its own database to manage. This allows for a greater scale of independency from team to team.
- **Fault resiliency:** The microservice mentality of "design for failure" brings the concept of "fault resilient application" naturally.
- **Built around business capabilities:** Allows for cross-functional team, or full-stack team. Every team has a full stack of competencies inside it.
- **Flexibility in technology and polyglotism:** Microservice can be built with different programming languages (the best for a given task), and still cooperate one another. The same can be said about technology.

CONS:

- **Communication overhead:** Every time a service needs to access another service resource, so in particular every time there is a server dependency, a communication cost has to be accounted for. The dependency graph in microservice based architectures can become a real mess, and the number of edges can explode pretty easily. This has to be considered when thinking about microservices.
- **Complexity:** The complexity is an exploding factor when dealing with a distributed environment, in particular if it resembles a "distributed monolith".
- **Wrong cuts:** the "goodness" of a microservice architecture resides on the good choices of the cutting points. If the software was cut on the wrong places, then the effect of this cut will result in an actual application performance degradation.
- **Avoiding data duplication:** Services will need to share data, and it is not always feasible to wrap these data in a service and provide an API, due to exceeding time/complexity cost. The solution, usually, resides in data duplication, which is a double edged knife, useful for microservice isolation, but risky in terms of consistency. Avoiding it as much as possible is a hard challenge.
- **A poor team results in a poor system:** Microservices are not easy, so the team needs to have the skills to handle them. In this environment (more than in the monolithic one), any product resembles the team that builds it.
- **Hard time monitoring:** while it might be easy to monitor a single microservice, it is not the case for the whole application.
- **Need for documentation:** With the growth of the number of service, there is a need for a centralized, well organized documentation.

WHICH ARE THE “SQUADS” AND “TRIBES” AT SPOTIFY? #LORENZO

Squads in Spotify are similar to the concept of teams in the microservice way of thinking. The squad has its own product owner, which handles and feeds the team with the user stories that have to be implemented. Every squad have a full-stack team and so have available all the needed tools for designing, developing, testing and releasing a service.
A tribe is a collection of squads within the same business area (for example all the squads working on parts of the mobile application). The squads of the same tribe sit near each other and periodically interact via shared lounges in order to improve cross-team communication. Even the tribes have their leader which has to make the tribe environment right.

WHAT IS A FAULT-INJECTION TESTING? #LORENZO

Fault Injection Testing (FIT) is a way to ensure the system works even in difficult conditions. It is a philosophy which is embraced by Netflix, which created tools like “Simian Army” in order to inject faults in the production environment (so reflecting on the actual user experience). The main idea is to stress the system by injecting faults either on the computation nodes (for example by shutting down some virtual machines running), or/and on the communication channels between services (by stressing the inter-service request mechanism). Developers are forced to write fault resilient code in order to “survive” these faults, resulting in an overall application capable of surviving over a whole data center being down without major repercussions on the users.

HOW CAN “DESIGN FOR FAILURE” BE ACTUALLY IMPLEMENTED? #LORENZO

A good example of an instance of design for failure implementation is the circuit breaker. The main concept of this approach is to insert a “middleware” between the client (in general, it might be a microservice calling another microservice), and the supplier. The main idea is to break the communication between those two entities by letting the circuit breaker handle errors in the supplier. In fact, if the client is actively waiting for a supplier response, and this entity is failing for any reason, then the client would starve potentially forever, leaving the user of the application waiting too. The circuit breaker sends requests to the supplier on behalf of the requesting entity and implements internally a timeout in order to decide whether the supplier is up and running or not. Of course the insertion of the circuit breaker brings implicitly a new entity, which is susceptible to failure too. The client (the requester) has to provide a timeout too in order to understand if the circuit breaker is down. The client can then show the user, in case of failure of either the circuit breaker or the supplier a “partial” response to his request. This response has to be perceived as a “non failure” as much as possible from the user by either showing a partial past result stored via caching or an error (might it be clear or not). One way to kind of “force” the programmers to develop fault aware code is to actually implement FIT (Fault Injection Testing) in order to force them to actually run code that runs and behaves over a faulty environment.

WHAT IS GIT/GITHUB? #ANDREA/#GDL

Git is a free and open source distributed version control system (VCS).
Version Control Systems are a category of software tools that help a software team manage changes to source code over time. A VCS keeps track of every modification to the code in a special kind of database, called index. If a mistake is made, developers can turn back the clock and compare previous versions of the code, in order to fix the errors in a quick way (rollback).

Despite other VCS like SVN, Git is not centralized and the history of the project (repository) is present in all nodes that have cloned it, so developers can work also when they are not connected to the server (or when they are offline). Only when modifications have to be published a connection to the shared repository is needed (or when you want to pull the most recent commits).

A commit stores the current contents of the index along with a log message from the user describing the changes. Commits are then pushed to the shared repository to made the edits available for everyone. The list of all commit of the repository compose the repository's history.

The well-known GitHub platform is a Web-based hosting service, for version control using Git technology. GitHub hosts a shared repository where one or more users can collaborate. Moreover, thanks to its plug-ins, in GitHub is possible to deliver software with CI/CD techniques. For instance, with CI tools you can run tests automatically, every time you push a new commit or make a pull request. Combined with CD tools, you can also test your code on multiple configurations, run additional performance tests, and automate every step until production. According to GitHub stats, the most used tools are Travis CI, Circle CI and Jenkins.

Continuous Delivery (CD)*: is a software engineering approach in which teams produce software in short cycles (eg. 2 weeks), ensuring that the software can be reliably released at any time. It aims at releasing software with greater speed and frequency to properly meet market requirements. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production.*

Continuous Integration (CI)*: is the practice of merging all developer working branches to a shared mainline several times a day. The longer a branch of code remains checked out, the greater the risk of multiple integration conflicts and failures when the developer branch is reintegrated into the main line. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes. Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell", where the time it takes to integrate exceeds the time it took to make their original changes.*

WHAT IS FLASK? #LORENZO

Flask is a Python (version 3.3 at least) microframework (micro in the sense that given framework takes a minimal set of decisions) which incorporates Jinja to render views. It was made to make the development of web applications faster. The Flask class inside the flask.app module is considered to be the entry point. Flask instantiates it and dispatches WSGI (web service gateway interface) requests to the right code.

WHAT IS THE MODEL-VIEW-CONTROLLER PATTERN? #LORENZO

The model-View-Controller pattern consists on three components:

- The Model, which is the entity that should manage the data.
- The view, which is the component that displays the model .
- The Controller, which manipulates the Model state.

The View, which is the one the client interacts with, catches user events and forwards them to the controller. The Controller then maps these events to state changes in the Model. The Model needs to inform the View of every change, and the view can query the model in order to retrieve state information. The controller can also select the view to be shown.

WHAT IS CELERY? #LORENZO

Celery is an open source asynchronous task queue, based on distributed message passing. It runs tasks on its own (on a timely basis). There is the need for an intermediary message broker (in our case redis), that passes datas fetched from celery to the application. Celery supports real time operation and scheduling.

WHAT IS / HOW CAN YOU USE OPENAPI 2.0? #ANDREA

The OpenAPI Specification is a community-driven open specification within the OpenAPI Initiative, a Linux Foundation Collaborative Project.
The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPI Specification removes guesswork in calling a service.
So, basically, an OpenAPI Specification file allow you to describe an API including:

- General information about the API
- Available paths (endpoints)
- Available operations on each path
- Input/Output data structure for each endpoint

An Open API file can be written either in JSON or YAML. This latter is far more easy to write and read than JSON, indeed YAML is the most used language. Even if an OpenAPI specification is a simple text file which can be edited with any text editor, it’s better to use a specialised tool like Swagger Editor.

While developing you can adopt a “spec first” approach, writing the API specification via OpenAPI and then implementing it in the app; or you can code your app’s endpoints and then extract them in a specification file.

SOFTWARE TESTING

WHAT IS DEVELOPMENT/ RELEASE/USER TESTING? #GDL

Development testing means that the system is tested during development to discover bugs and defects. Testing might include static code analysis, peer code reviews, code coverage analysis. The minimal amount of tests that should be done is:

- unit testing: testing program components, such as methods or object classes
- component testing: several units integrated to create composite components, testing interfaces
- system testing: components integrated, system tested as a whole, testing component interactions

Release testing is a form of system testing where a separate team tests a complete version of the system before it gets released to users. The aim is checking that system meets requirements of system stakeholders, to convince the system supplier that it is good enough for use. Testing might include end to end or functional tests.

User testing means that a restricted number of real users test the system in their own environment. It is essential since user's real working environment cannot be fully replicated but it can impact on reliability/performance/usability of the system. The types of user testing are:

- alpha testing: users work with development team to test early releases of software
- beta testing: release made available to larger group of users to allow them to experiment and to raise problems that they discover with the system developers
- acceptance testing: customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer's environment

WHAT IS PARTITION TESTING? #GDL

There are two main strategy to compare when we talk about unit testing effectiveness:
- partition testing: identify groups of inputs that should be processed the same and choose tests from within each of these groups
- guideline-based testing: choose test cases by using testing guidelines (that reflect previous experience of the kinds of errors that programmers often make when developing components)

In partitioning testing we identify equivalence partitions – classes of input (and output) data that have «common characteristics» (e.g., positive/negative numbers, menu selections) and for which program normally behave in a comparable way, then we create test cases from each partition in order to test boundaries input to our functionalities.

WHAT IS TEST-DRIVEN DEVELOPMENT? #GDL

Test-driven development, or TDD, was introduced as part of agile methods. It is an approach to program development in which you inter-leave testing and code development. With this approach, tests are written before code, and the fact the the code «passes» the tests is the critical driver of development. Only when this fact occurs a new test can be introduced and the iteration repeats. In this way the code is added incrementally to the tests, giving developers more confidence before deployment. A TDD cycle is composed by:

1. Identify the increment of functionality that is required (normally small and implementable in a few lines of code)
2. Write a test for this functionality and implement this as an automated test
3. Run the test, along with all other tests that have been implemented
4. Initially, you have not implemented the functionality so the new test will fail
5. Implement the functionality and re-run the test
6. Refactor properly the code and run the tests again
7. Once all tests run successfully, you move on to implementing the next chunk of functionality

PROS:

- code coverage: every code segment that you write has at least one associated test so all code written has at least one test
- avoids regression testing (testing the system to check that changes have not broken previously working code)
- simplified debugging: when a test fails, it should be obvious where the problem lies
- system documentation: the tests themselves are a form of documentation that describe what the code should be doing

WHAT IS LOAD TESTING? #GDL

Performance testing is meant to ensure that software applications will perform well under their expected workload. A software application's performance like its response time, reliability, resource usage and scalability do matter. The goal of Performance Testing is not to find bugs but to eliminate performance bottlenecks.
The types of performance testing are:

- **load testing**: checks the application's ability to perform under anticipated user loads. The objective is to identify performance bottlenecks before the software application goes live.
- **stress testing**: involves testing an application under extreme workloads to see how it handles high traffic or data processing. The objective is to identify the breaking point of an application.
- **endurance testing**: is done to make sure the software can handle the expected load over a long period of time.
- **spike testing**: tests the software's reaction to sudden large spikes in the load generated by users.
- **volume testing**: under volume testing large number of data is populated in a database and the overall software system's behavior is monitored. The objective is to check software application's performance under varying database volumes.
- **scalability testing**: the objective of scalability testing is to determine the software application's effectiveness in "scaling up" to support an increase in user load. It helps plan capacity addition to your software system.

USER STORIES

WHAT IS AN USER STORY FOR? #GDL

A user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system, and they define what has to be built in the software project. User stories are prioritized by the customer (or by the product owner in Scrum) to indicate which are most important for the system and will be broken down into tasks and estimated by the developers. When user stories are about to be implemented, the developers should have the possibility to talk to the customer about it. Short stories may be difficult to interpret, they may require some background knowledge or the requirements might have changed since they were written. Every user story must at some point have one or more acceptance tests attached, allowing the developer to test when the user story is done and also allowing the customer to validate it. Without a precise formulation of the requirements, prolonged non constructive arguments might arise when the product have to be delivered.

WHAT IS THE PRIORITY OF A USER STORY? #GDL, LORENZO

The priority of a user story is assigned by project stakeholders. It represents the importance of an activity compared to others. Generally speaking, the highest is the priority for a story, the more important its value-proposition is in the business model of the application. Usually, top priority stories get implemented as soon as possible by the developing team in order to have a functional application skeleton (proof of concept) to show to users in order to verify requirements. The main idea behind this fact is that having an actual application skeleton to show to the user can help him understand what is the direction of the work, and it gets easier for him to create additional user stories, change some of the previous according to the work that has been done.

WHAT IS THE EFFORT/SIZE OF A USER STORY? #GDL

The effort of a user story is the estimated amount of time that developers will use to completely implement such story. This estimation is done by developers themself.
The size of a user story is another viewpoint of the effort, since they are strictly related.

WHAT IS AN EPIC? #GDL

Epics are large user stories, typically ones which are too big to implement in a single iteration and therefore they need to be disaggregated into smaller user stories at some point.
Epics are typically lower priority user stories because once the epic works, its way towards the top of the work item stack it is reorganized into smaller ones. It doesn't make sense to disaggregate a low-priority epic because you'd be investing time on something which you may never get to addressing, unless a portion of the epic is high priority and needs to be teased out.

SPLITTING THE MONOLITH

WHEN AND WHERE TO START SPLITTING THE MONOLITH CODEBASE? #LORENZO

It should be splitted in the moment in which handling a strongly coupled architecture (which means redeploying the whole monolith every time a small update to the code is made) and non-cohesiveness (all sorts of code together), becomes problematic. Splitting should start from the "most beneficial" one, where most beneficial is according to the splitting team. Typical drivers would be Pace of Change (most susceptible to change soon), Team Structure (from a geographical point of view), Security (the most critical one), Technology (services based on tools, algorithms…), Tangled dependencies (the least dependent one from others).

To properly define how the codebase should be splitted, developers can highlight seams of the project to identify service boundaries. This process can exploit, for example, the concept of namespace of programming languages to find margins that must, subsequently, be redefined.

HOW TO SPLIT DATABASES? (E.G. HOW TO BREAK FOREIGN KEY RELATIONSHIP?) #LORENZO

To split the database we first have to identify the code lines that read/write on the db, then understand where are the foreign key relations. To break foreign key relations, the "pointed" db should be wrapped in a service, which is supposed to expose an API to retrieve/update the internal db. To share static data you should wrap them into a service, or duplicate the static table.

WHAT ABOUT TRANSACTION WHEN YOU SPLIT? #LORENZO

The solutions to transaction's problem are:

- In case of error, abort and compensate (prob: where to handle compensation? What if compensation fails?)
- Distributed transactions, with voting and transaction manager (prob: single point of failure in the transaction manager, blocking approach, overhead, what if commit fails after vote?)
- Eventual consistency: Client determines when data is considered consistent.

WHAT IS THE SAGA PATTERN? #GDL

The SAGA is a design pattern used to maintain data consistency across services. Each transaction that spans over multiple service must be implemented as a saga. A saga is a controller that manages a sequence of transactions. Each transaction updates the database and publishes a message (event) to trigger the next transaction in the saga. If a transaction fails, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding transactions.

There are two ways of coordination sagas:

- Choreography: each transaction publishes events that trigger transactions in other services
- Orchestration: an orchestrator (the saga controller) tells the participants what transactions to execute

PROS:

- It enables an application to maintain data consistency across multiple services without using distributed transactions

CONS:

- The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.
- In order to be reliable, a service must atomically update its database and publish a message/event.

WHAT IS EVENTUAL CONSISTENCY? #GDL

Eventual consistency is a characteristic of distributed computing systems such that the value for a specific data item will, given enough time without updates, be consistent across all nodes. Before enough time has elapsed without updates, however, the value may be inconsistent across multiple copies of the data. Eventual consistency is only acceptable for fault-tolerant applications.
Eventual consistency is also known as “optimistic replication”, a strategy for replication in which replicas are allowed to diverge.

As a simple example, if an application contains a way of viewing some part of the database state, and a way of editing it, then users may edit that state but then not see it changing in the viewer. Alarmed that their edit "didn't work", they may try it again, potentially more than once. If the updates are not idempotent (e.g., they increment a value), this can lead to disaster. Even if they are idempotent, the updates place a overhead on the database system - and the situation in which replication delays become particularly noticeable is when the database system is at a high level of load.

WHAT IS A (EVENT) DATA PUMP? #LORENZO

Data pumps are related to the reporting system, which generally needs data from multiple databases. In a distributed context, as it is the case when handling microservices, the service itself periodically pushes data to the report service using the so called data pumps (built by the service team, not the report one). Data pumps can also piggyback on other backup operations. Event data pumps are a pattern according to which the service generates events based on internal state changes. The report system can, in this case, subscribe to this events to receive data, reducing in this way the binding.

WHAT ARE THE SERVICE/DEPLOYMENT MODELS OF CLOUD COMPUTING? #ANDREA

Service models

- **Software as a Service (SaaS)**: is a model who encloses applications and software system, accessible from any type of device through the use of a simple client interface. In this way, the user does not have to worry about managing the resources and the infrastructure, because they are managed by the provider.
  Examples of SaaS: Google Docs, Office 365, ecc..

- **Infrastructure as a Service (IaaS)**: is the first level of the cloud, used for access, monitoring and management of remote infrastructures, such as virtualized computers, storage, networking and network services (such as the firewall). The customer therefore, does not have to worry about the actual hardware but "rents" a certain amount of space, cpu and ram needed for their own purposes. Although the user is free to install the platform he prefers, all the software operability is in his charge.
  Example of IaaS: Amazon Simple Storage Service (S3), Google Cloud Storage, ecc..

- **Platform as a Service (PaaS)**: PaaS services are in the mixture of Iaas and SaaS. Platform-as-a-Service solutions deliver scalable and elastic runtime environments on demand and host the execution of applications. These services are backed by a core middleware platform that is responsible for creating the abstract environment where applications are deployed and executed. It is the responsibility of the service provider to provide scalability and to manage fault tolerance, while users are requested to focus on the logic of the application developed by leveraging the provider's APIs and libraries. This approach increases the level of abstraction at which cloud computing is leveraged but also constrains the user in a more controlled environment.
  Example of PaaS: Microsoft Azure, Heroku, ecc..

Deployment models

- **Private Cloud**: A private cloud consists of computing resources used exclusively by one business or organization. The private cloud can be physically located at your organization's on-site datacenter, or it can be hosted by a third-party service provider. But in a private cloud, the services and infrastructure are always maintained on a private network and the hardware and software are dedicated solely to your organization. In this way, a private cloud can make it easier for an organization to customize its resources to meet specific IT requirements. Private clouds are often used by government agencies, financial institutions, any other mid to large-size organizations with business-critical operations seeking enhanced control over their environment.

- **Public Cloud**: Public clouds are the most common way of deploying cloud computing. The cloud resources (like servers and storage) are owned and operated by a third-party cloud service provider and delivered over the Internet. With a public cloud, all hardware, software, and other supporting infrastructure is owned and managed by the cloud provider. In a public cloud, you share the same hardware, storage, and network devices with other organizations or cloud "tenants." You access services and manage your account using a web browser. Public cloud deployments are frequently used to provide web-based email, online office applications, storage, and testing and development environments.

- **Hybrid Cloud**: Hybrid clouds combine private clouds, with public clouds so organizations can exploit the advantages of both. In a hybrid cloud, data and applications can move between private and public clouds for greater flexibility and more deployment options. For instance, you can use the public cloud for high-volume, lower-security needs such as web-based email, and the private cloud (or other on-premises infrastructure) for sensitive, business-critical operations like financial reporting.

AN EXAMPLE OF (DISRUPTIVE) BUSINESS MODEL EXPLOITING CLOUD COMPUTING? #ANDREA,LORENZO

An example of a disruptive business model would be Dropbox. Its business model, and its disruptive idea (let's offer free storage to everyone) was only possible thanks to the low cost of the AWS S3 buckets used by Dropbox to store the actual data of the users. The main idea is: Dropbox had only a handful of datacenters, in which it stored the user metadatas, while the actual data were hosted on AWS thanks to its very accessible cost. Dropbox then proposed two kind of offers to its client: one free, with fixed storage and some limitations, and a premium one, with less limitations and a file collaboration feature which was better than Amazon's. The revenue stream (which was only the one from premium users) had to be enough to:

- Pay Amazon for both premium and free users.
- Pay its own data centers.
- Pay the salaries.

This was only allowed by the cheap cost of cloud storage resources.

WHAT IS A VIRTUAL MACHINE/ SERVER VIRTUALIZATION? #ANDREA #LEONARDO

Virtualization is a technology that allows creation of different computing environments (software stacks). It contains a collection of solutions allowing the abstraction of some of the fundamental elements for computing, such as hardware, runtime environments, storage, and networking. Virtualization has become a fundamental element of cloud computing. This is particularly true for solutions that provide IT infrastructure on demand, in fact, virtualization confers that degree of customization and control that makes cloud computing appealing for users and sustainable for cloud services providers.

The most common example of virtualization is hardware virtualization (both computation, storage and network). This technology allows simulating the hardware interface expected by an operating system. Hardware virtualization allows the coexistence of different software stacks on top of the same hardware. These stacks are contained inside virtual machine instances, which operate in complete isolation from each other, and are managed by a hypervisor. High-performance servers can host several virtual machine instances, thus creating the opportunity to have a customized software stack on demand. This is the base technology that enables cloud computing solutions to deliver virtual servers on demand (Amazon EC2, VMware vCloud, and others).

WHAT ARE HEROKU'S DYNOS FOR? #ANDREA

Dynos are simply lightweight Linux containers dedicated to running your application processes. At the most basic level, a newly deployed app to Heroku will be supported by one Dyno for running web processes. You then have the option of adding additional Dynos and specifying Dyno processes in your "procfile". Dynos actually come in three different flavors:

- **Web Dynos**: for handling web processes
- **Worker Dynos**: for handling any type of process you declare (like background jobs)
- **One-off Dynos**: for handling one-off tasks, such as database migrations.

One of the great things about Heroku Dynos is how easy they are to scale up and out. Through Heroku's admin portal or via the command line, you can easily add more Dynos or larger Dynos. Adding additional Dynos can help speed up your application's response time by handling more concurrent requests, whereas adding larger Dynos can provide additional RAM for your application.

Operating-system-level virtualization, also known as **containerization**, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances. Such instances, called **containers**, partitions, virtual environments (VEs) or jails (FreeBSD jail or chroot jail), may look like real computers from the point of view of programs running in them. A computer program running on an ordinary operating system can see all resources (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container. On Unix-like operating systems, this feature can be seen as an advanced implementation of the standard chroot mechanism, which changes the apparent root folder for the current running process and its children. In addition to isolation mechanisms, the kernel often provides resource-management features to limit the impact of one container's activities on other containers. System-level-virtualization is frequently implemented in remote access applications with dynamic cloud access, allowing for simultaneous two-way data streaming over closed networks.

WHAT IS FAAS? #ANDREA

Function as a service (FaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications. AWS Lambda, was the first FaaS offering by a large public cloud vendor, followed by Google Cloud Functions, Microsoft Azure Functions, ecc..
Use cases for FaaS are associated with "on-demand" functionality that enables the supporting infrastructure to be powered down and not incurring charges when not in use. Examples include data processing (e.g., batch processing, stream processing, ecc..), Internet of things (IoT) services for Internet-connected devices, mobile applications, and web applications.
Differently from PaaS, where developers must care about scalability, in FaaS, everything is managed by the providers, in this way, developers may focus only on the code, rather than monitoring the services. Moreover, in those systems, developers usually pay only for function execution time (and no process idle time), so, they can achieve larger scalability at a lower price.

Amazon Lambda motto: "Just upload your code and Lambda takes care of everything required to run and scale your code with high availability."

HOW TO AVOID/REDUCE CLOUD LOCK-IN? #ANDREA, #GDL

There are some steps that ease to avoid vendor lock-in:

- Carefully choose the provider, and negotiate both an entry and exit strategy
- Have a backup vendor and keep on-premise options open
- Prefer open standard and open-source software
- Avoid to use provider's add-ons unless they are really needed (eg. for time to market reasons)
- Ensure portability of data for a future migration
- Design decoupled portable applications (API/REST integration), such as microservices architectures
- Use containers and DevOps tools

WHAT IS A CONTAINER/IMAGE/VOLUME? #LORENZO

A container is a lightweight virtualization mechanism that exploits kernel isolation features like chroot and cgroups. A container's purpose is to offer a lighter, faster to start, simpler to build and to deploy virtualized environment to deploy applications on. The developer should only worry about packaging the code via providing its sources and its dependencies, and the container is then able to run anywhere, without the developer burden of managing a virtual machine.
Images are read only templates which are exploited to create and run containers. An image is a collection of filesystem layers and some metadata. Metadata offer some informations like port mappings (which virtual container port is mapped to which physical server port), and environment variables. The layers are modified version of the application, and a new layer can be added via the command commit (in a Docker environment). A container lifespan is bound to the existence of the main process of the application. The moment this process stop, the container dies, and every data inside it is lost, as a container is a volatile abstraction. Volumes are the Docker solution for this problems, and they can be created via the command *docker volume create*. Volumes are an "outside of the container file system" directory, handled by Docker, and that can be read/written by the container in order to make some data persistent.

WHAT IS THE DIFFERENCE BETWEEN A VIRTUAL MACHINE AND A CONTAINER? #ANDREA

A **Virtual Machine (VM)** can be described as a software program that emulates the functionality of a physical hardware or computing system. It runs on top of an emulating software called the hypervisor, which replicates the functionality of the underlying physical hardware resources with a software environment. These resources may be referred to as the Host Machine, while the VM that runs on the hypervisor is often called a Guest Machine. The virtual machine contains all necessary elements to run the apps, including the computing, storage, memory, networking and hardware functionality available as a virtualized system. It may also contain the necessary system binaries and libraries to run the apps, while the OS is managed and executed using the hypervisor. The virtualized hardware resources are pooled together and made available to the apps running on the VM. An abstraction layer is created to decouple the apps from the underlying physical infrastructure. As a result, the physical hardware can be changed, upgraded or scaled without disrupting the app performance. A VM will operate as an isolated PC and the underlying hardware can operate multiple independent, isolated VMs for different workloads.

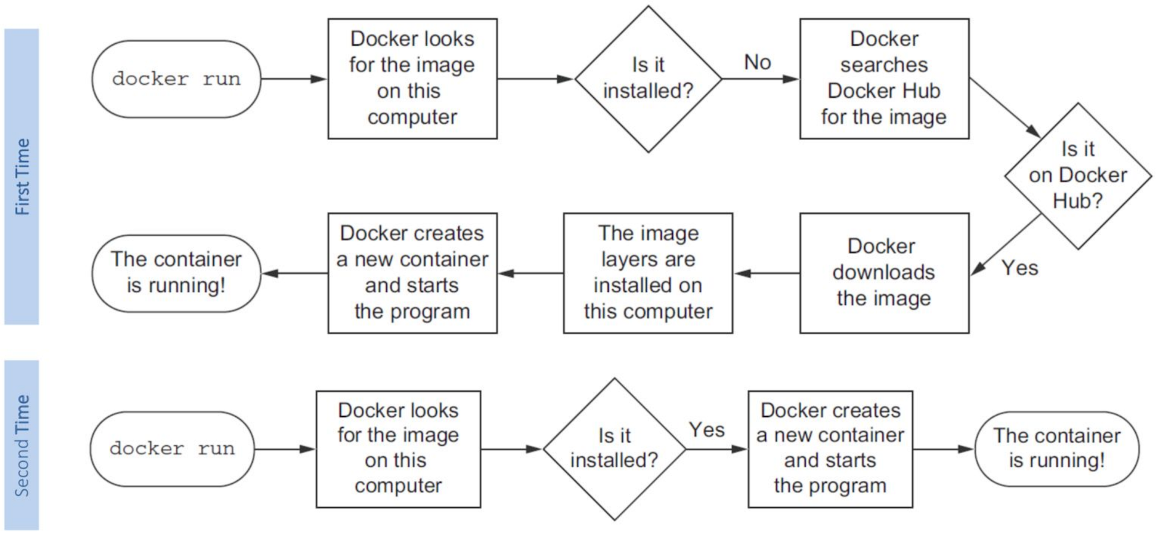
Containerization creates abstraction at an OS level that allows individual, modular and distinct functionality of the app to run independently. As a result, several isolated workloads can dynamically operate using the same physical resources. Containers can run on top of bare metal servers, hypervisors, or in the cloud infrastructure. They share all necessary capabilities with the VM to operate as an isolated OS environment for a modular app functionality with one key difference. Using a containerization engine such as the Docker Engine, containers create several isolated OS environments within the same host system kernel, which can be shared with other containers dedicated to run different functions of the app. Only bins, libraries and other runtime components are developed or executed separately for each container, which makes them more resource efficient as compared to VMs. Containers are particularly useful in developing, deployment and testing of modern distributed apps and microservices that can operate in isolated execution environments on same host machines. With containerization, developers don’t need to write application code into different VMs operating different app components to retrieve compute, storage and networking resources. A complete application component can be executed in its entirety within its isolated environment without affecting other app components or software.

WHAT IS IMAGE LAYERING IN DOCKER? #LORENZO

Image layering is a mechanism used by Docker in order to reduce the disk space of images (imagine the case of a lot of copies of an image running in different containers, if everyone needs its files somewhere on the disk then the disk space explodes). The main idea is to “use” the fact that a Docker image is made of read-only layer. Every time a docker container gets instantiated, only the image files goes actually on the file system, and a copy on write mechanism is activated, so anytime a container tries to modify a page, a page fault is raised, and a new page is allocated on the file system in which the container has the right to write. The fast starting mechanism of container is made possible by this thing (layer caching), since all the files are already in the filesystem (because of the image), then after docker run, not much data has to actually be allocated. Docker commit then comes easy. Any time a docker commit command is done, then the layer corresponding to the last container version is frozen, and labeled with its own identifier.

WHAT IS THE EFFECT OF DOCKER RUN/COMMIT? #ANDREA

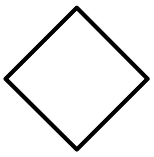
Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When an operator executes “docker run”, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.


First Time
docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer
Docker creates a new container and starts the program
The container is running!
Second Time
docker run
Docker looks for the image on this computer
Is it installed?
Yes
Docker creates a new container and starts the program
The container is running!

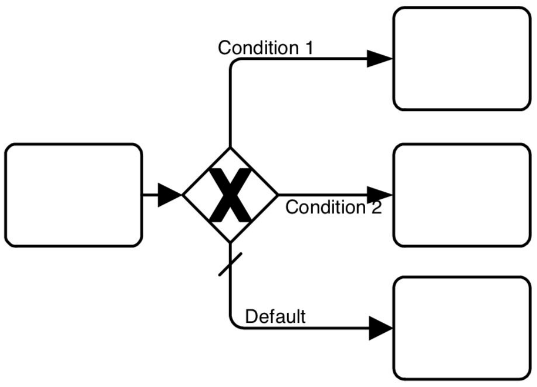
The “docker commit” command allows users to take a running container and save its current state as an image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server. Generally, it is better to use Dockerfiles to manage images in a documented and maintainable way. The “docker commit” operation will not include any data contained in volumes mounted inside the container.

WHAT IS A PARALLEL/EXCLUSIVE/INCLUSIVE GATEWAY IN BPMN? #ANDREA

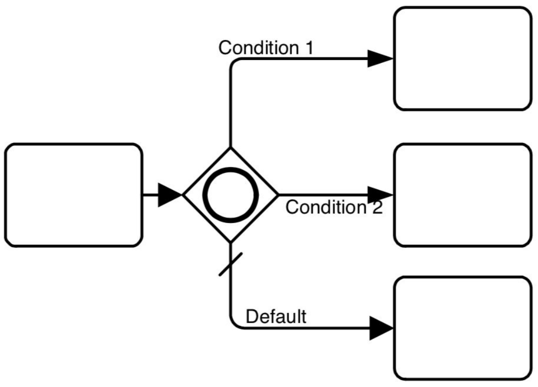
Gateways are used to control how Sequence Flows interact as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term “gateway” implies that there is a gating mechanism that either allows or disallows passage through the gateway that is, as tokens arrive at a gateway, they can be merged together on input and/or split apart on output as the gateway mechanisms are invoked. A gateway is a diamond that must be drawn with a single thin line



Exclusive gateway:
An Exclusive Gateway (Decision) is used to create alternative paths within a Process flow. For a given instance of the Process, only one of the paths can be taken. A Decision can be thought of as a question that is asked at a particular point in the Process. The question has a defined set of alternative answers. Each answer is associated with a condition Expression that is associated with a Gateway’s outgoing Sequence Flows.The Exclusive Gateway may use (not mandatory) a marker that is shaped like an “X” and is placed within the Gateway diamond to distinguish it from other Gateways.



Inclusive gateway:
A diverging Inclusive Gateway (Inclusive Decision) can be used to create alternative but also parallel paths within a Process flow. Unlike the Exclusive Gateway, all condition Expressions are evaluated. The true evaluation of one condition Expression does not exclude the evaluation of other condition Expressions. All Sequence Flows with a true evaluation will be traversed by a token. Since each path is considered to be independent, all combinations of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken. The Inclusive Gateway must use a marker that is in the shape of a circle or an “O” and is placed within the Gateway diamond to distinguish it from other Gateways.



Parallel gateway:
A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows. The Parallel Gateway must use a marker that is in the shape of a plus sign and is placed within the Gateway diamond to distinguish it from other Gateways. A Parallel Gateway creates parallel paths without checking any conditions; each outgoing Sequence Flow receives a token upon execution of this Gateway. For incoming flows, the Parallel Gateway will wait for all incoming flows before triggering the flow through its outgoing Sequence Flows.

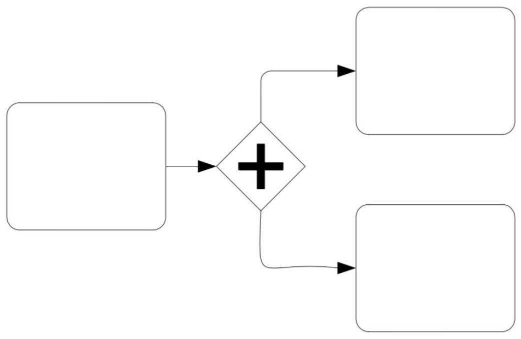


Figure 10.110 - An example using an Parallel Gateway

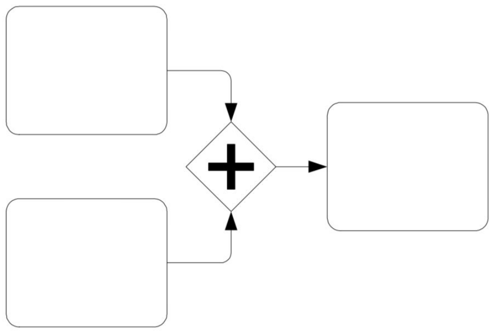
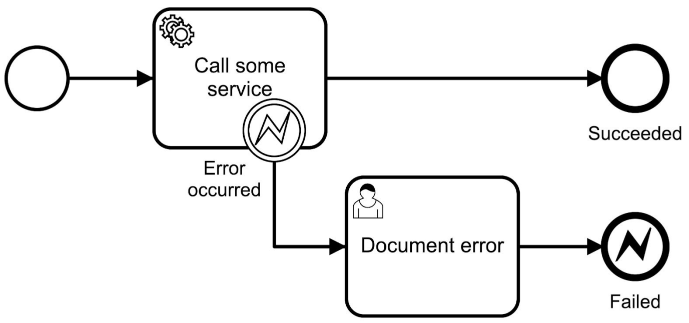


Figure 10.111 - An example of a synchronizing Parallel Gateway

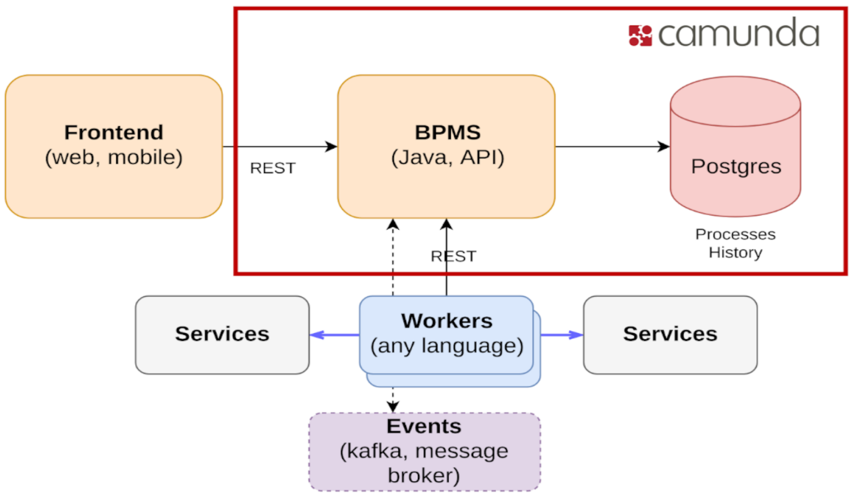
WHAT IS A ERROR EVENT IN BPMN? #ANDREA, #GDL

Error events are events which are triggered by a defined error. A BPMN error is meant for business errors, which are different than technical exceptions. So, this is different than Java exceptions - which are, by default, handled in their own way. An error event attached to the boundary of an activity (either a task or a subprocess) will catch an error if that activity is being processed when the error is thrown. If the error is thrown after the activity has been completed, then it will not be caught.



WHAT IS CAMUNDA? #LEONARDO

Camunda is a framework supporting BPMN for workflow and process automation. Workflows are defined in BPMN which can be graphically modeled using the Camunda Modeler. It provides a RESTful API which allows you to use your language of choice.



CAN YOU DESCRIBE THE USAGE PATTERNS OF CAMUNDA?

You start by modeling graphically your BPM adding some “execution ordering’” arrows, events, gateways and tasks. Then you can deploy the BPM to the camunda framework (it can be installed locally or runned inside apache tomcat in a docker container). Once the model has been deployed you can execute it via the camunda web frontend. While the process is executing the various tasks are completed. We saw three kinds of this tasks:

- **Expression Service Task:** when the flow goes through this task an expression is evaluated and it’s result can be assigned to a process variable.
- **User Task:** this task requires the user interaction that is done via a form. Depending on the form it can be filled by the user or just displayed with some process variables values.
- **External Service Task:** this task is the most interesting since it’s execution is handled by an external RESTful worker. Each tasks of this kind provide a unit of work (it is the process engine that actually creates the external task instance) to a topic queue. This queue can be polled by some external worker (i.e. a NodeJS worker) which fetches, locks and completes the task.

WHAT IS THE DIFFERENCE BETWEEN ORCHESTRATION AND CHOREOGRAPHY? #LEONARDO

Choreography:
Service choreography permits to services to self-coordinate in a P2P fashion. Some examples of choreography could be a traffic roundabout or a group of dancers.
Orchestration:
Orchestration is the centralised and automated coordination of services. As examples of orchestration we could mention traffic lights or a band (orchestra).

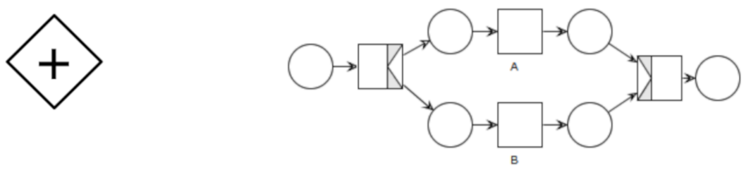
WHAT IS A WORKFLOW NET? #LEONARDO

Workflow nets are petri-net-based workflows that is suitable for expressing workflows. A Petri net is one of several mathematical modeling languages for the description of distributed systems.

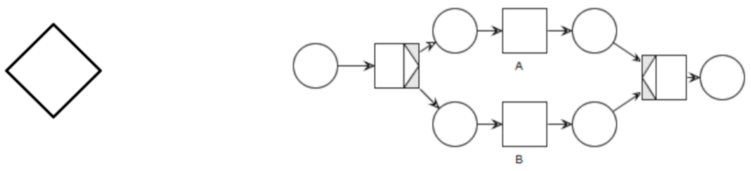
A workflow net is a petri net iff:

1. There is an initial place with no incoming edges
2. There is a final place with no outcoming edges
3. All places and transitions are located on some path from the initial place to the final place

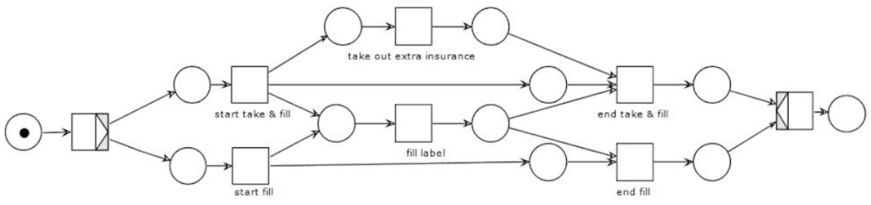
HOW CAN WE MODEL BPMN PARALLEL/EXCLUSIVE/INCLUSIVE GATEWAYS WITH WORKFLOW NETS?

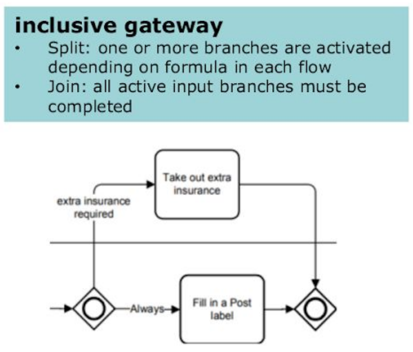


The first one is a parallel gateway, while the second it's an exclusive (xor) gateway.



In the second picture below an example of inclusive gateway is represented. Be careful that this example fits only the case shown in the picture where one branch of the gateway is "always" executed; if the branch conditions are different this kind of implementation may not work anymore.





inclusive gateway

- Split: one or more branches are activated depending on formula in each flow
- Join: all active input branches must be completed

WHAT IS A LIVE/BOUNDED/SOUND NET? #LEONARDO

A workflow net is sound iff:

1. (bound) every net execution starting from the initial marking (one token in the initial place, no tokens elsewhere) eventually leads to the final marking (one token in the final place, no tokens elsewhere)
2. (live) every transition occurs in at least one net execution

A Petri net is **live** iff for every reachable state M' and every transition t, there is a state M" reachable from M' which enables t.
A Petri net is **bounded** iff for each place p there is a natural number n such that for every reachable state the number of tokens in p is less than n.

A workflow net N is sound iff (Ň,i) is live and bounded.

FOG COMPUTING

WHAT IS FOG COMPUTING? #LORENZO

Fog computing is a computing paradigm that was proposed in order to overcome the emerging limitations of Internet of Things. IoT allows for new interesting applications in everyday life, like domotics, automatic transport, drone fleet and drone delivery, visual security (through CCTV camera surveillance), smart cities and farms… The main problem is that there are a lot of IoT devices, and a lot more are expected to join the Internet. In addition to that, the amount of data generated by each and every one of these devices is huge, and the cloud is not expected to hold the IoT momentum. In addition to that, some of the presented applications, have some precise QoS requirements (like latency, offline capability…) that cannot be provided in an only cloud based interaction. The idea of fog computing is the following: fill the gap between data centers and IoT with one (or more) intermediate computation layer populated by fog nodes, which act as filters towards the data centers (in order to reduce the amount of data sent to the cloud), and a physically close endpoint for latency sensitive applications. This way, the best of the cloud deployment model (virtually infinite resources) is kept, and the best of the edge deployment model (where everything is handled locally) is also kept (freedom of choice, low latency, complete control).The main points that are allowed by this paradigm are:

- Low Latencies and bandwidth savings
- Geo-Distribution
- Collaboration (between fog nodes)
- Heterogeneity of devices
- Mobility support
- Location and Context awareness

WHAT IS / HOW DIFFICULT IS THE “COMPONENT DEPLOYMENT PROBLEM” IN FOG COMPUTING? #LORENZO

Given

- a multi-component application A with requirements R
- a distributed Fog infrastructure I (nodes and links)
- a set of objective metrics M

determine eligible application deployments that meet all requirements in R and optimise metrics in M .
This problem is known as Component Deployment Model.

The component deployment problem (CDP) is an NP-Hard problem, which means that a NP-complete problem can be reduced to CDP in polynomial time (so it is a problem which is at least as hard as an NP complete one, but it might also fall outside of NP). In particolar, the proposed proof of NP-hardness was by reduction from SIP (Subgraph Isomorphism Problem), which is known to be NP-Complete. The reduction is made by making SUPER-graphs vertices into Fog nodes with available resources set to 1, the edges of the SUPER-graph into links between nodes with bandwidth equal to 1, the vertices of the SUB-graph into the various application components with resource requirements set to 1 and the edges of this SUB-graph into component interactions with bandwidth requirements equal to 1. This SIP < CDP reduction were performed in polynomial time so CDP is NP-Hard.

WHAT IS FOGTORCHPI? #LORENZO

Fog-Torch-PI is an open source prototype developed by the SOCC (Service-Oriented, Cloud and Fog Computing research group of Pisa’s computer science department) that solves CDP (component deployment problem) via backtracking.
It takes into account non-functional parameters within the model (i.e., hardware, software, latency and bandwidth) to determine, compare and contrast different eligible deployments of a given application over a Fog infrastructure.
In the case of hardware capabilities, it considers CPU cores, RAM and storage available at a given node or required by a given software component.
Software capabilities are represented by a list of software names (operating system, programming languages, frameworks etc).
It considers latency, and both download and upload bandwidths as QoS attributes. Latency is measured in milliseconds (ms), while bandwidth is given in Megabits per second (Mbps).

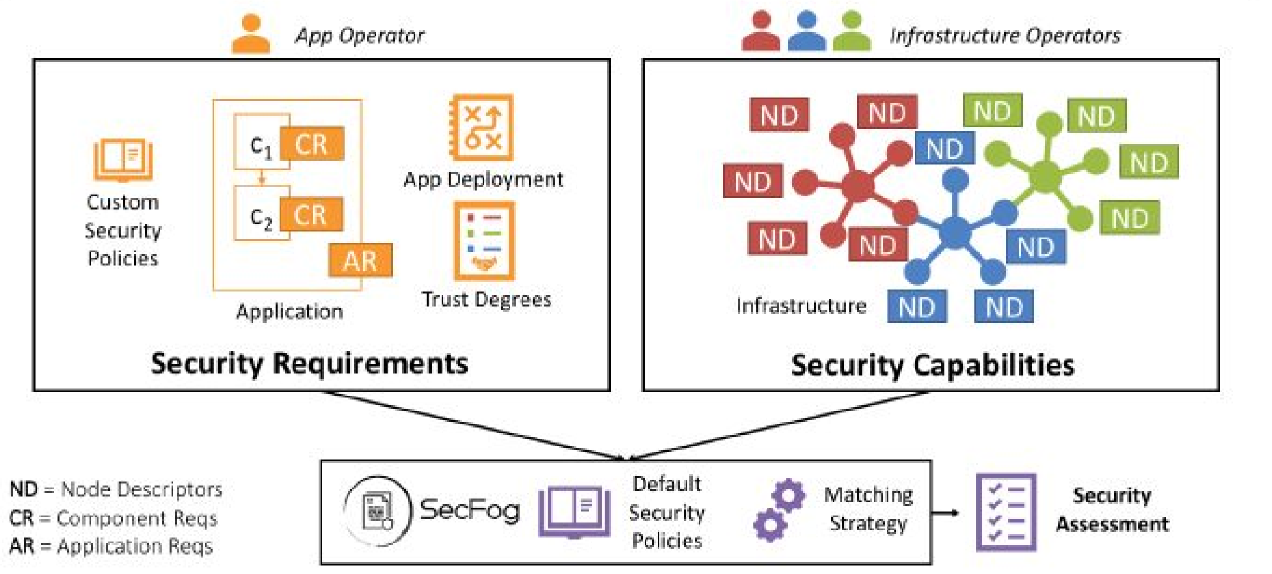
FogTorchPI estimates the QoS of the links through Monte-Carlo simulation in order to address the user towards the best deployment regarding 3 dimensions: cost, fog resource consumption and QoS assurance.

The Monte-Carlo simulation is needed because the QoS of nodes and links is not static in reality, but it could vary according to a certain probability distribution.

The main features of FogTorchPI are:

- QoS awareness: deployments with less average QoS should be valued less than the ones with more
- location awareness: data are processed near the place where they are created
- context awareness: fog nodes should cooperate with one another

SecFog is a declarative methodology proposed in order to assess the security level of a multi component application deployment in Fog scenarios (in practise, it fills the security gap that was missing in the FogTorchPI, which totally relied on other functional and non-functional properties.
The main idea is to first propose a taxonomy of security terms based on the actual security needs of fog computing (this proposal is needed since no standard control frameworks are available for a fog environment). SecFog then takes the informations (trust degrees) regarding the infrastructure, and tries to match the user's security policies in order to retrieve a security assessment.



The main idea would be to use FogTorchPI output together with SecFog in order to combine the 3-dimensions obtained by FogTorchPI (resource consumption, cost and QoS assurance) and merge its result with the security assessment proposed by SecFog for given deployment(s), and this can be obtained via Multi-Objective optimisation by ranking those 4 dimensions according to user's priority.