Advanced Software Engineering

University of Pisa

A.Y. 2018-2019

Professors: Antonio Brogi, Stefano Forti

Summary by Daniele Gadler

Core Interoperability Standards

1. What is Rest?

REST (REpresentational State Transfer) is an abstract model for the architectural structuring of *HTTP* (HyperText Transfer Protocol) and *URIs* (Uniform Resource Identifiers). HTTP is the main protocol utilized for the exchange of requests / responses between clients and a server over the internet for updating the client state. URIs are a way to locate and address a resource. In REST, a resource also identifies a service exposed to the client.

In brief, REST suggests the ***structuring of a REST web-based application*** as a Virtual-State-Machine. The server exposes a set of resources, and the client navigates through these resources, keeping track of the state on his local end. Whenever a hyperlink is clicked or a form submitted, and the user lands on a new page, the local state changes through a message exchange based on HTTP. Messages exchanged can be of different types, i.e: XML or JSON. JSON is an unstandardized message exchange protocol used for simple applications, whereas XML is a well-standardized hierarchical information structuring protocol, allowing for the creation of more complex applications.

In REST, the ***state*** is uniquely client-based, as the server stores no information about the client's information – If the server were to do this, then this approach would be totally unfeasible and unscalable. Therefore, we deem this a "distributed representation of the server's state in each client accessing the server's services". Furthermore, REST is generally ***lightweight*** and ***scalable*** thanks to the light-weight components it leverages, such as: HTTP for communication, which is a stateless protocol. Because of the sheer amount of requests that a web service would need to be facing daily, ***efficiency*** and ***scalability*** are key factors that have fostered the development of REST and its widespread adoption for the creation of simple webservices.

REST Principles

We have seen that there exist 4 main principles for the creation of RESTful services, namely:

1. *Resource Identification through URIs:* A RESTful service exposes a set of resources – hence services – through URIs, which are accessible by clients. Such URIs constitute a global addressing space for service accessibility and discovery.
2. *Uniform Interface:* Resources are manipulated through 4 well-defined operations:
   a. ***PUT****:* Create a new resource (Ex. create a new order on Amazon).
   b. ***GET:*** Retrieve the current state of a certain resource. (Ex.: view the state of your order on Amazon)
   c. ***POST:*** Update the state of a certain resource (Ex.: Update your billing method on Amazon): the client transfers a new state to a server resource. NB: This is not a violation of bullet 4 of the present list, as the client just has the new resource state locally and transfers it to the server via a *non-idempotent request*.

d. ***DELETE:*** Delete a certain resource (Ex.: Cancel an order because you already got the same object as a Christmas present).

3. *Self-Descriptive Messages:* REST is based on self-contained messages, meaning that requests contain enough context information to process the whole message. It is *"message-centric"* [notes] and *"resource-centric"*[slides], as the only way to update a resource is via messages exchanged via request-responses between a client and a server. Also, resources (EX: resources hosted on a webservice programmed in Python) are decoupled from their representation, which can occur via several different standards (EX: the representation of a webservice programmed in python is an HTML page embedding JSON objects). Metadata about a certain resource retrieved can be used to control caching (EX: Pictures generated in Matplotlib over a local server folder with the same name are saved in the damn Google Chrome cache and not refreshed whenever being generated anew, as the writer experienced).
4. *Stateful Interactions through hyperlinks*: <u>Every interaction with a resource is stateless.</u> Session state is stored uniquely on the client-side. When the client wants to change its local state, it will need to perform some requests to the server, requesting its local state to be changed. (EX: To order a book, the state required for the user is 'logged-in' The client will then need to go through a series of hyperlinks and/or forms to change its local state to 'logged-in' before ordering a book).

   N.B: GET requests are idempotent and can be repeated with no consequences on the server's state, whereas POST requests are not idempotent, as they update the server's state of a certain resource to the local client's state of that very resource (performing a state transfer).

2. How can we create/update/access resources in REST?

Creating, updating, deleting, accessing resources occurs via the four operations listed in the REST principles for question [1]. The following is an example taken from the slides, providing a high-level description of such process.

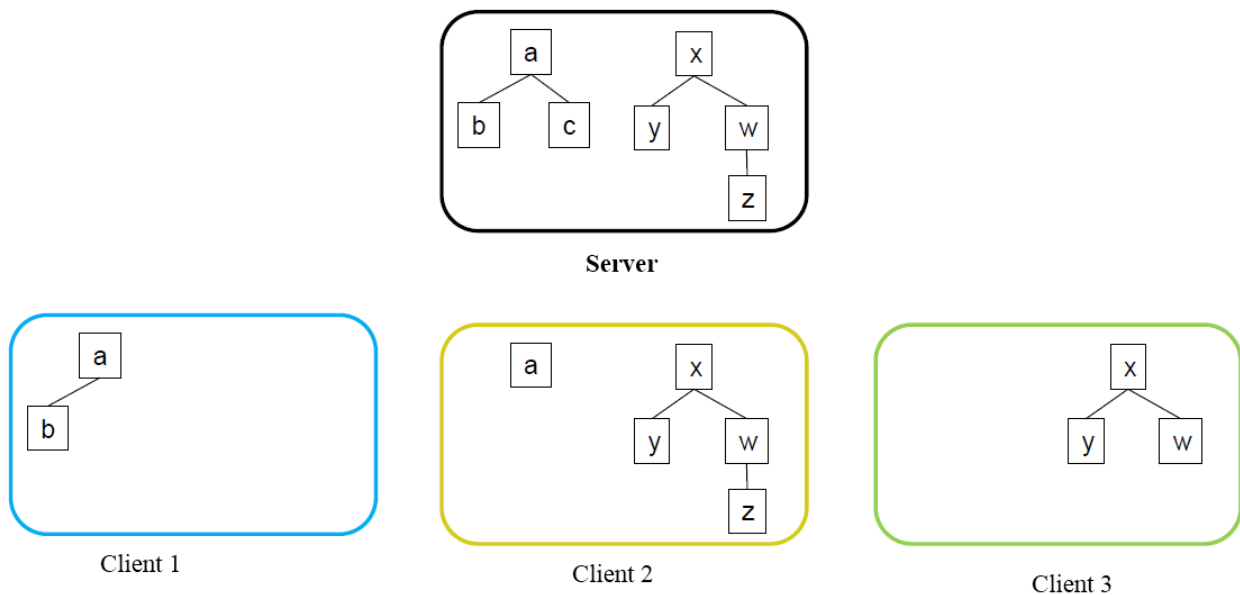


Figure 1: Colored rectangles represent clients, whereas the black rectangle represents the server. The nodes and edges inside the rectangles represent the local view of the webserver that each client has (a REpresentation of the server's state), as accessed by a client via GET requests.

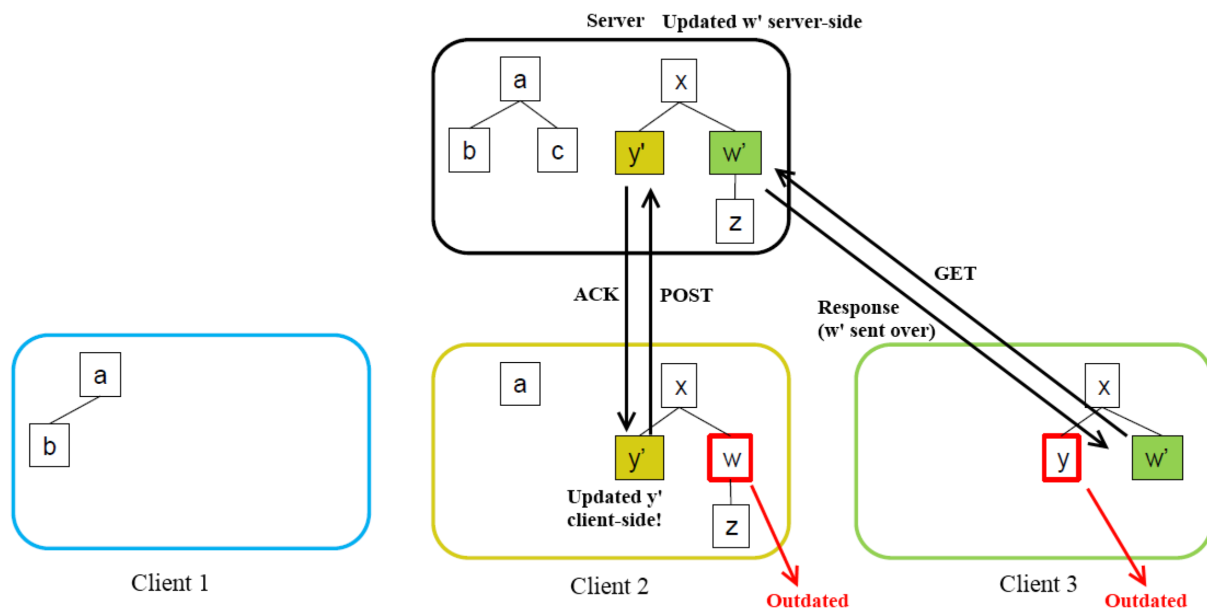
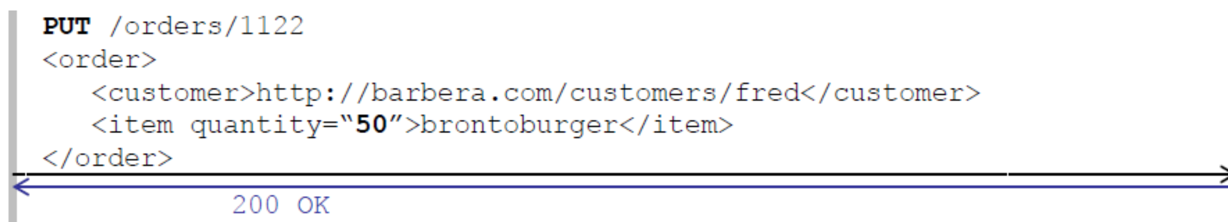


Figure 2 The same legend of Figure 1 applies. Step (1): Client 2 updates y to y' locally and send a POST request to the server, receiving an ACK back confirming the POST's successful completion. Step (2): Server updated w to w'. Step (3): Client 3 performs a GET request for w and gets w' back.

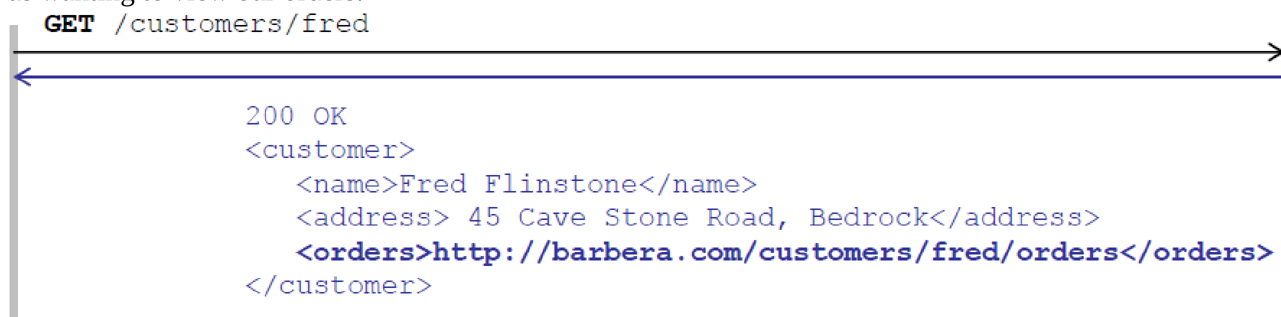
- **Create a resource:** We create a resource by performing a PUT request. For example, let's consider the case of us wanting to place a new order to get a Brontoburger delivered to us:



```
PUT /orders/1122
<order>
   <customer>http://barbera.com/customers/fred</customer>
   <item quantity="50">brontoburger</item>
</order>
              200 OK
```

Figure 3 We perform a PUT request by sending an XML message encapsulating sufficient information needed by the server to create our order. Then, we get an ACK response with the "200" response code back, signifying that the request was successfully completed. Recall that messages need to be self-contained to be processed by the server. This is because HTTP is a stateless protocol.

- **Access a resource:** We access a resource by performing a GET request. For example, in the case of us wanting to view our orders:



```
GET /customers/fred

              200 OK
              <customer>
                 <name>Fred Flinstone</name>
                 <address> 45 Cave Stone Road, Bedrock</address>
                 <orders>http://barbera.com/customers/fred/orders</orders>
              </customer>
```

Figure 4 We perform a GET request by specifying the URI of the resource we wish to access. The server's response is the response code indicating the state of the performed request, followed by an XML message encapsulating the information desired.

- **Update a resource:** We update a resource by performing a POST request. For example, in the case of us casting a vote in a recently created poll with ID "112233", analogously to Assignment (1).

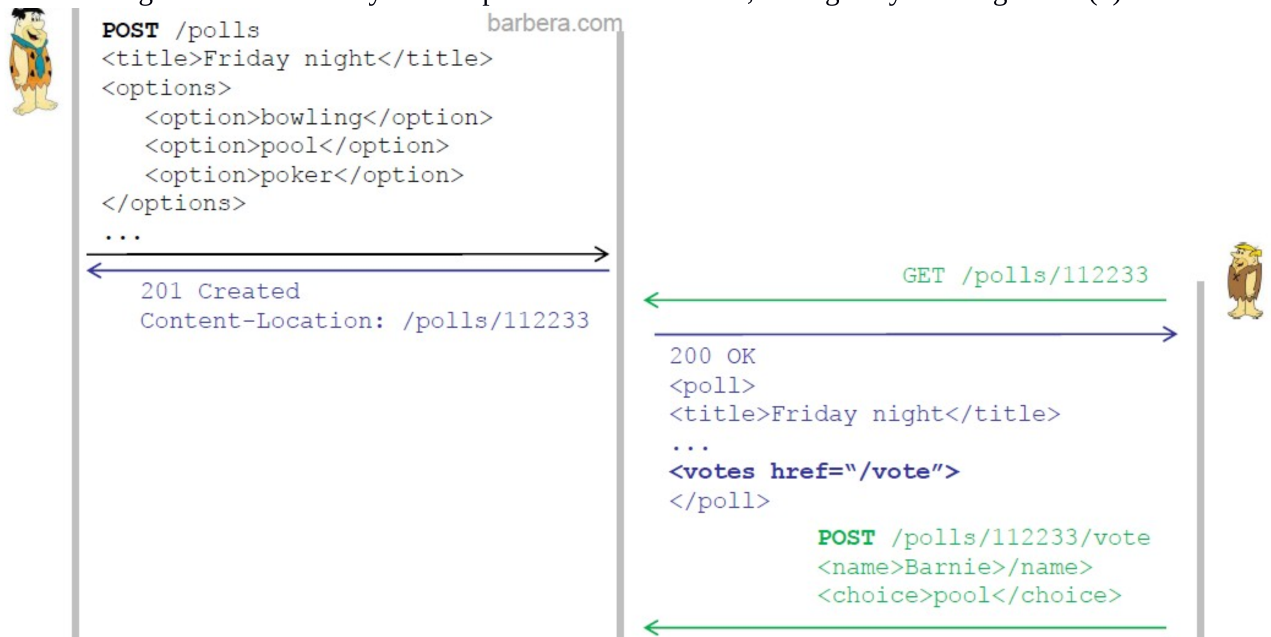


Figure 5: Barney and Fred are the two clients. barbera.com is the webserver. Step (1): Fred performs a POST request to create a new poll (lo-REST). The server then returns a 201 response code, confirming the successful poll creation. Step (2): Barney first performs a GET request to view the newly created poll with ID "112233", then performs a POST request to cast a vote and specifying the parameters to be set.

- **Delete** a resource: We delete a resource by indicating the URI we wish to delete. The response is a response code indicating whether the URI specified did indeed exist and could hence be deleted successfully.

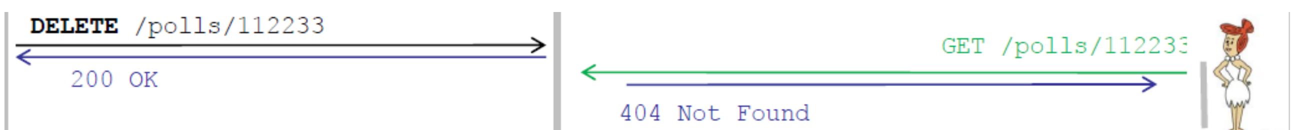


Figure 6: We perform a DELETE request analogously to a PUT request, specifying the URI of the resource we wish to delete. The response is then a status code specifying the status of the performed request.

3. Which are the pros and cons of REST?

REST is a relatively young standard born in the early 2000s', allowing to build lightweight web-services from scratch with minimal customization and specification. It is based on widespread and ***well-established standards*** (HTTP, XML, URIs), which are intelligible and parsable by web-browsers. It also features good scalability and efficiency characteristics. However, it comprises its simplicity with lacky support for QoS, security and capacity of encoding complex data. Also, some proxies and firewalls uniquely support POST and GET request, with POST requests being used for all non-idempotent operations: POST, DELETE and GET requests being used for all idempotent operations: GET, PUT. Consequently, this has led to the creation of some dirty workarounds, through which the actual verb is sent via the "X-HTTP-Method-Override" field of the request header. These workarounds are in no way standards though, and may lead to compatibility issues in certain web browsers.

Disadvantages	Advantages
Too simple to model complex situations	Simple to use and learn

Apparently trivial design decisions may have serious impacts at a later stage	Efficient and scalable, based on the internet’s core protocol for message exchange (HTTP)
Confusion in the adoption of Hi-REST and Lo-REST. Dirty workarounds for compatibility.	Deployment and testing made easy, as no client code needs to be coded, but just the server code.
No best practices mandated in the design of REST-APIs	Re-use of existing client infrastructure (server browser)
Limitations in the encoding of very long and complex data structures (Ex: special/Asian characters) in URLs	Flexible usage of layers (firewall, proxies)

4. What is SOAP?

Simple Object Access Protocol is an XML-based standard relying on HTTP as a transport protocol for exchanging information one-way between separate systems running on heterogeneous infrastructures. We can hence consider it an ***interoperability standard***, as it is mainly used for allowing communication between separate applications, abstracting over the different applications’ programming languages and mechanisms.

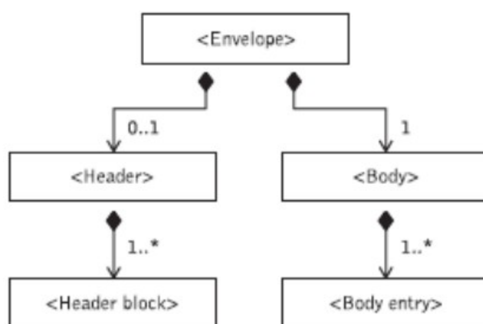
SOAP defines how to format and process communication messages exchanged between HTTP-based URLs (endpoints). In order to send a message to a service S, the sender must know:

- the *names* and *parameters* of the operations supported by service S,
- the *address* of S
- the *transport protocol* to be used

WSDL comes in handy to provide a through description of these three main elements for a web service.

SOAP message

A SOAP message consists of a SOAP Envelope, which wraps the XML document contained therein and provides a way to augment the payload with some further information used for routing.

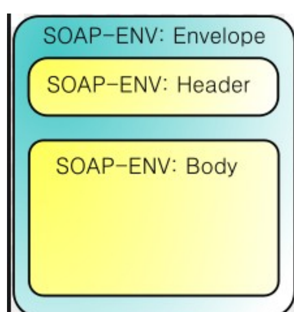



```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
   xmlns:env="http://www.w3.org/2003/05/soap-envelope">

   <env:Header> <!-- optional -->
    <!-- header blocks go here . . . -->
   </env:Header>

   <env:Body>
    <!-- payload or Fault element goes here . . . -->
   </env:Body>
</env:Envelope>
```

Figure 7: Pictorial representation of a SOAP envelope and its content



[SOAP Header] - Optional: Contains metadata about HOW the soap message must be processed and, , like an IP packet travelling over several different hops, contains a header that stores information related to the sender address and the destination address, as well as some control bits.

The SOAP header also features a SOAP message sender, a SOAP message recipient and eventually a digital signature. This is done to allow SOAP intermediary hops to process the SOAP message and forward it over the next hop correctly.

When receiving a SOAP message with a SOAP header, a node checks the “role” attribute: only nodes acting in that role will be authorized to process that SOAP message (ex: if the role found is “ultimateReceiver”, then only the last hop will be authorized to process that SOAP message).

SOAP Body – Mandatory: It contains either-application specific payload, or a fault message. Not both of them! One single fault block can be carried in a SOAP envelope. A fault message is arisen in case an error occurs during processing of the SOAP message. After opening the envelope received, the body can finally be parsed: the body generally contains the operations that the client wants to perform and in consequence of which a request to the server to carry them out is sent;

5. How are SOAP messages transported?

SOAP Communication Styles

SOAP messages are transported over HTTP as a transport protocol. Furthermore, SOAP supports two main communication styles:

a. *RPC, Remote Procedure Call*: The SOAP body contains an element with the name of the method or operation being invoked. An RPC-Styled webservice appears as a remote object to a client application invoking it.

```
<env:Envelope
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
         <tx:Transaction-id
            xmlns:tx="http://www.transaction.com/transaction"
              env:mustUnderstand='1'>
                512
         </tx:Transaction-id>
    </env:Header>
    <env:Body>
         <m:GetProductPrice>
            <product-id>  450R6OP  </product-id>
         </m:GetProductPrice >
    </env:Body>
</env:Envelope>
```

Figure 8: SOAP RPC message request, requesting the product price of product with ID 450R6OP

```
<env:Envelope
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
         ...
    </env:Header>
    <env:Body>
         <m:GetProductPriceResponse>
            <product-price>   134.32  </product-price>
         </m:GetProductPriceResponse>
    </env:Body>
</env:Envelope>
```

Figure 9 SOAP RPC message response, containing the price of the product requested

b. *Document-Style.* The SOAP Body contains one or more child elements called *parts.* It contains whatever the sender and the receiver agree upon.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
  ...
  </env:Header>
  <env:Body>
    <po:PurchaseOrder oderDate="2004-12-02" xmlns:po="http://www.plastics_supply.com/POs">
          <po:from>
             <po:accountName>   RightPlastics   </po:accountName>
             <po:accountNumber>   PSC-0343-02  </po:accountNumber>
          </po:from>
          <po:to>
              <po:supplierName>  Plastic Supplies Inc.  </po:supplierName>
              <po:supplierAddress>  Yara Valley Melbourne  </po:supplierAddress>
          </po:to>
          <po:product>
              <po:product-name> injection molder </po:product-name>
              <po:product-model> G-100T </po:product-model>
              <po:quantity> 2 </po:quantity>
          </po:product>
     </ po:PurchaseOrder >
  </env:Body>
</env:Envelope>
```

Figure 10: Document-Style SOAP message

SOAP messages encoding

encoding style

communication style	Literal	Encoded
RPC		
Document		

SOAP messages are encoded in two main ways when being transported:

a) *Literal*: data is serialized according to a schema, which is usually expressed using the W3C XML schema.
b) *SOAP Encoded*: Set of serialization rules that specify how objects, structures, arrays are encoded for transport.

SOAP messaging patterns

Soap messages can be transmitted in a *bidirectional synchronous fashion* by sending two one-way asynchronous messages, meaning that the client or server waits for a response to be received before proceeding with further requests or solicits

- Client Server. Server Client. ***Request-Response***
- Server Client. Client Server. ***Solicit-Response***

Alternatively, messages can be transmitted in *a one-directional asynchronous fashion*, meaning that the client or server receiving the message sends no response back upon receiving a message.

- One-way: Client Server
- Server Client: notification

```
POST /Purchase Order HTTP/1.1
Host: http://www.plastics_supply.com
Content-Type:application/soap+xml;
charset = "utf-8"
Content-Length: …

<?xml version="1.0" ?>
<env:Envelope
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
        <tx:Transaction-id
            xmlns:tx="http://www.transaction.com/transactions"
            env:mustUnderstand='1'>
                512
        </tx:Transaction-id>
    </env:Header>
    <env:Body>
        <m:GetProductPrice>
            <product-id>  450R6OP  </product-id >
        </m:GetProductPrice >
    </env:Body>
</env:Envelope>
```

Figure 11: SOAP Request message over HTTP. A POST request is sent to place an order for the item with ID 450R6OP

```
HTTP/1.1 200 OK
Content-Type:application/soap+xml;
charset = "utf-8"
Content-Length: ...

<?xml version="1.0" ?>
<env:Envelope
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
       <--! - Optional context information -->
    </env:Header>
    <env:Body>
       <m:GetProductPriceResponse>
          <product-price>   134.32  </product-price>
       </m:GetProductPriceResponse>
    </env:Body>
</env:Envelope>
```

Figure 12: HTTP SOAP message response to the request sent in Figure 11

6. What is WSDL?

The Web Service Definition Language (WSDL) is a platform- and language- independent XML-based language used to describe the *public interface* of a service. Whereas SOAP mandates how to format and process messages exchanged, WSDL is used to specify *what* a service does, *where* it resides and *how* to invoke it by allowing us to describe a service S via:

- ***What***: the *names* and *parameters* of the operations supported by service S
- ***Where***: the *address* of S (URI of S)
- ***How***: the *transport protocol* to be used for communication

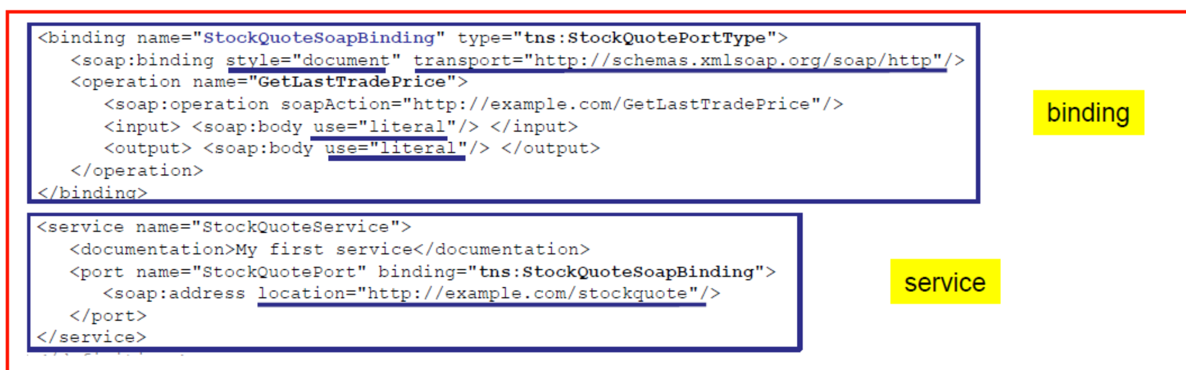
WSDL hence provides the establishment of a ***service contract*** between a client (requester) and a server (provider), in form of an interface.

Parts of a WSDL Interface:

1. ***Abstract service description**:* It is an abstract description of the *operations* and their *parameters*, and *abstract data types* supported by service S. A service S can support multiple bindings from multiple addresses and ports to the same abstract interface. The WSDL description contains all the info a requester needs to communicate with the service provider.

Figure 13: Example of an abstract interface, description of the interface Types, message structures into different parts and operations grouped by portTypes

2. ***Concrete endpoint implementation***: Performs an actual **binding** of an abstract interface by setting a networking address and port with a specific protocol. Still, WSDL bindings do not contain neither any programming language code nor service-specific implementation details. The concrete endpoint implementation implements the abstract interface by specifying:
   - A `binding`, which defines how a `portType` and `operations` (defined in an abstract interface) are mapped into a concrete representation by specifying:
     - *Concrete protocols:* Ex: SOAP over HTTP, SOAP over SMTP
     - *Messaging Styles:* RPC or document-style
     - *Encoding Styles:* Literal or SOAP encoding
   - A `service`, which binds a service to a specific net location by specifying one or more ports. Analogously to standard *socket ports*, a *port* is used to distinguish multiple service implementations



Concrete Endpoint Implementation

Figure 14: Concrete implementation of the abstract interface of Figure 13 by specifying a Binding, where the "document-style" message-passing scheme is specified, literal encoding is used, and HTTP is used as a transport protocol. The service location associated with the service implementation is also specified.

Following are the main differences between a RESTful architecture and a standard Web-Service (SOAP)-based architecture

	REST	**WS*-SOAP**
Learning curve	Lean(low gradient)	Steep(high gradient)
Layering	Flexible, free. Any #layers	Complex layering
Programming	Low-Level	Middle/High-Level
Features' lead time	Short	Long
QoS Support	No QoS support (best effort)	Strong QoS guarantees
Messages	Brief, human-intelligible	Long, machine-intelligible
Applications' architecture	Simple	Complex
Main usage	Mashups, simple webservices	Enterprise applications
Communication Protocol	HTTP	HTTP

7.What is a request-response operation in WSDL?

WSDL supports four different operation types and any combination of them. A *message exchange pattern* is defined based on the possible order and the presence of of the input and output elements supported by a certain operation.

- A ***Request-Response message exchange*** is a two-way message exchange mechanism, where the sender sends a SOAP request. Through this paradigm communication is *synchronous*: the sender is blocked waiting for a SOAP response from the receiver. Upon finally receiving the awaited SOAP response, the sender "unblocks" and is left free to perform further requests. During this process, the `service` receives requests and sends responses back. Generally, operations performed in this manner are Remote Procedure Calls, which request remote objects for local usage. Both an *input* and an expected *output* are specified.

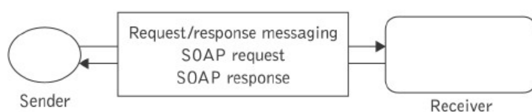


Figure 16: Request-Response message Exchange paradigms

```
<operation name="GetLastTradePrice">
   <input message="tns:GetLastTradePriceInput"/>
   <output message="tns:GetLastTradePriceOutput"/>
</operation>
```

Figure 15: Request-Response operation specification with both an input and an output

- **One-Way messaging**: The service endpoint receives the message, but does not send a response back. This paradigm works in an asynchronous fashion; In the operation, just an *input* is specified and no *output*, as no response is awaited from the receiver.
[Simple Request paradigm, with no response]

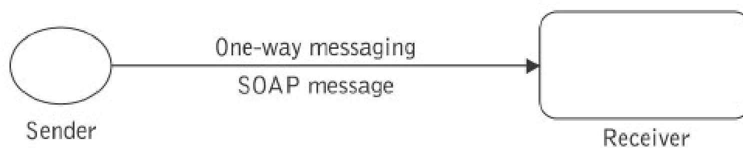


Figure 17: One-Way messaging message exchange

- **Notification operations**: This paradigm is analogous to the publish-subscribe paradigm. Clients subscribe to the service, which then sends a notification to all subscribed clients. The service does not await a response from the clients. Only an `output` element is specified and no input.

Figure 18: Notification messaging message exchange

- **Solicit-Response**: This is just the opposite of the Request-Response paradigm. The server performs a request to the client, and it is then blocked waiting for a response from the client. The operation declares an `output`, followed by an `input` element.

Figure 19: Solicit-Response message exchange

Single Message Passing	One-Way Message	One-way messaging SOAP message Sender Receiver	*The operation can receive a message but will not return a response.*
	Notification	Notification messaging SOAP notification Sender Receiver	*The operation can send a message but will not wait for a response.*
Two-way Message Exchange	Request/Response	Request/response messaging SOAP request SOAP response Sender Receiver	*The operation can receive a request and will return a response.*
	Solicit/Reply	Solicit/response messaging SOAP request message SOAP answer message Sender Receiver	*The operation can send a request and will wait for a response.*

8.What is a portType / port in WSDL?

A `portType` is specified in an ***Abstract Interface Description***; it is a uniqueID that defines a group of abstract operations – analogously to an interface in Java (actually in WSDL 2.0 portTypes were renamed to interfaces). Operations in a portType are grouped by the portType specified.

```
<portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
   </operation>
</portType>
```

featured operation(s), grouped in portType(s)

Figure 20: Set of operations grouped by portType in an Abstract Interface Description. Notice that we define the abstract "GetLastTradePrice" operation in the "StockQuotePortType" portType.

A `port` is specified in a ***Concrete interface Description***; It is a *binding* of a single endpoint with a single address. In WSDL 2.0, the *port* was renamed to *binding*.

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input> <soap:body use="literal"/> </input>
      <output> <soap:body use="literal"/> </output>
   </operation>
</binding>
<service name="StockQuoteService">
   <documentation>My first service</documentation>
   <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
   </port>
</service>
```

binding

service

Concrete Endpoint Implementation

Figure 21: Concrete implementation of the "GetLastPrice" abstract operation for the "StockQuotePortType" portType , defined in an abstract way in Figure 15; In the concrete implementation, we have the association of the endpoint "http://exampole.com/stockquote" with the binding "tns:StockQuoteSoapBinding" for the "GetLastTradePrice" operation.

Figure 22: Summary of the Abstract Interface and Concrete Interface's components

Extra – WSDL usage

There exist tools that convert WSDL into programming languages, both for client-side code and server-side service code (EX: Generate stubs – methods with an empty body – reflecting the interface specified in WSDL)

Proxy classes: client-side images of (remote) object classes that implement the web service. The client-side application invokes the proxy, which forwards the invocation to the actual remote service. The proxy performs marshalling and unmarshalling of arguments.

Microservices

1. Which are the main characteristics of microservice-based Architectures?

The main reasons for a microservice-based architecture are the following ones:

a. *Reduced lead time*: In order to adapt to an ever-changing market scenario, companies need to get code in production as quickly as possible. Microservices dramatically help to reduce the lead time from the design of a new functionality to the first user click in production. This is achieved by automatizing and streamlining the deployment process and achieving independent microservice deployment. Microservices are built around business capabilities by cross-functional teams, and chords are reduced.

b. *Horizontal scaling:* Whereas in monolithic applications the only possible way to scale an application is by replicating the whole application (*vertical scaling*), in microservices-based applications, horizontal scaling is actually made possible by uniquely replicating the components that represent the bottleneck (critical part) of the application.

Figure 23: Horizontal scaling of a single component of an application

c. *Overcome the main issues of a monolithic application*. A monolithic application is made up of different parts that need to interact with one another. The critical part is especially represented by the server-side backend, where one single tiny little change would cause the whole application to be re-built and re-deployed, leading to an increased lead time. In a monolithic application, scaling is only possible in a vertical manner, and not in a horizontal fashion, as by bullet (b). In monolithic applications, the quality of code also tends to decay largely over time and maintenance costs consequently increase.

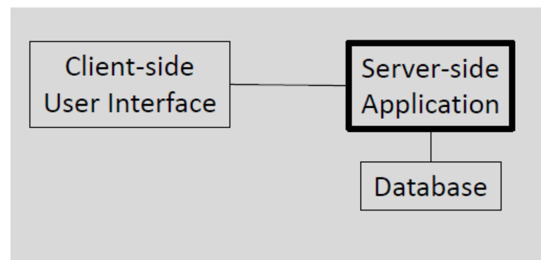
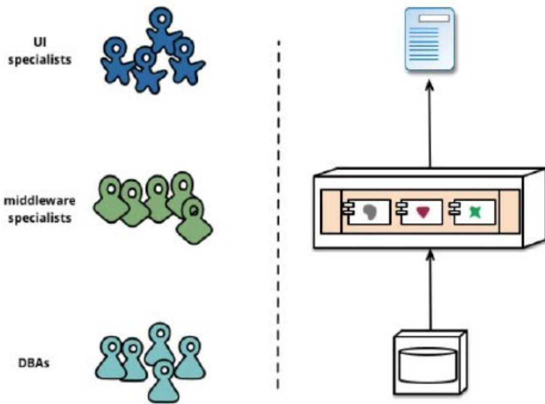


Figure 24: Architecture of a monolithic application

The main ***characteristics*** of a microservice-based architecture are the following ones:

1. ***Service Orientation***: An application is made up of a set of independently developed and deployable services, hence representing a real implementation of a *SOA* (Service-Oriented Architecture) "made right". Each service runs potentially multiple processes in its own container and interacts with other services via lightweight mechanisms, based on the REST protocol and according to the principle *"dump pipes, smart endpoints"*: no complex logic enforcing security or policies lies in the bus handling service-to-service communication as in ESBs (Enterprise Serial Bus), but the logic lies in the endpoints wanting to communicate. Each service hence exposes a set of functionalities made available by a REST API, and other services may use functionalities of other services via HTTP request-responses.
   Communication is *lightweight*, as the infrastructure is typically dumb, and communication is generally asynchronous: RabbitMQ or ZeroMQ just provide a way to achieve asynchronous communication.
   A change in one service just requires redeployment of that very service and doesn't affect other services until the service interface is changed; also, services are *polyglot*, in a sense that they can be coded in different programming languages, which are capable of communicating by means of an agreed common standard (i.e: HTTP request-responses).
   All of these design principles were actually the ones upon which the **Unix** architecture was built.
2. ***Organize services around Business Capabilities***: According to Conway's law, we have that:
   "Organizations that produce systems are constrained to produce designs which are copies of the communication structures of these organizations". *Conway's Law* – 1968
   In pre-agile software engineering organizations, teams used to be split into functional teams (i.e: "Database Admins' team", "Backend programmers' team") ..: such organizations were doomed to produce an application organized in siloed functional layers, adding significant communication burden functional teams because of the *chords* existing between different functional teams. A chord is a communication taking place between two different teams, and adds significant delay to the lead time.



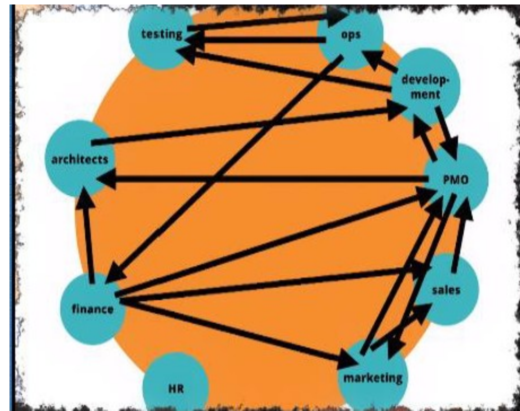


Figure 25: Monolithic application built by three separate functional teams results in an application split into three separate functional parts, as by Conway's Law

Figure 26: Chords in an organization split into functional teams.

On the other hand, the microservice approach to features' division is different, as services are split based on business capabilities. With microservices, we would have *cross-functional hybrid teams*, responsible for the different business features of the product (EX: One team would have one DB administrator, one front-end developer and a backend developer, and it would be responsible for the "Orders" feature). Such organizational structuring would hence imply a software architecture to be organized around business capabilities; if this restructuring is well implemented and, teams would be working on different independent parts of the code, thus leading to a dramatic reduction in the amount of chords within an organization.

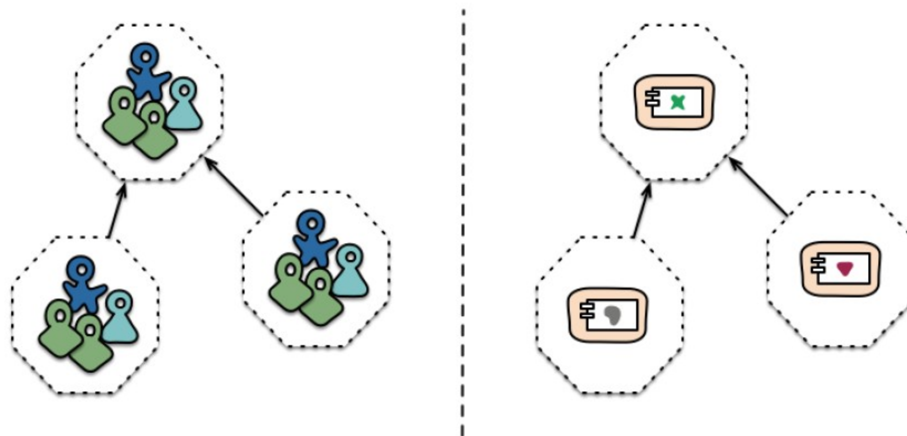


Figure 26: Software architecture organized around business capabilities resulting from cross-functional teams

EX: Hospital organized in a cross-functional way based on the illness to cure with reduced chords,

3. ***Decentralized Data Management***:

As applications are no longer organized in functional silos, each microservice is going to have its own "in-house" database. Decisions related to database design are hence decentralized from a team of database experts to the team managing the microservice. Although this decentralization helps reduce the chords in an organization - as the single database no longer represents a contention point among different teams - On the other hand this may lead to redundant information being stored in different

microservices' databases, and data consistency needing to be enforced. The only way to enforce consistency among different databases is via distributed transactions. Different consistency models can be enforced in distributed databases, however they all impose significant temporal coupling – which may significantly impact the application in a detrimental effect.

Consequently, microservices emphasize *transactionless* coordination among services, recognizing that consistency may only be *eventual consistency*, and possible issues arising are dealt with by compensating operations. (EX: Amazon Customer placing an order, yet still not seeing it among the orders because the distributed DBs are updating). Such trade-off is worth it as long as the cost of compensating transactions is less than the cost of the lost business under greater consistency standards.

4. ***Independently deployable Services***: Services can be deployed independently one from one another via CI (Continuous Integration, EX: We made use of the Travis CI Tool) and Continuous Delivery (CD). Releases are no longer "events", but are made "boring". This is a key factor in updating, extending and restarting microservices in a seamless way. We hence implement the concept of *evolutionary design*, with an emphasis on independent replaceability, upgradeability and maintenance.
   Also, we recognize that some services are more important than others and need "special care" (EX: Netflix's API Gateway: if it goes down, the whole Netflix application would be made unavailable and these are scaled in a horizontal manner.

5. **Horizontally Scalable Services:** Thanks to a microservice architecture, we are able to scale one particular part of an application (i.e: one microservice) simply by dedicating more resources to it, in case it represents a bottleneck by replicating that very microservice via a *farm*.

Figure 27: Farm handling of a critical component requiring more resources

6. **Fault-Resilient Services**: We accept the fact that system will constantly fail, and we adopt a proper design to make our application fault-resilient, in order to avoid one single failure leading to a cascade of failures. More on this in Question (4).

2. What are the main pros and cons of microservices?

Pros	Cons
Reduced chords thanks to cross-functional teams	Increased monitoring complexity because of the larger amount of smaller application parts
Integration, testing and delivery automatized thanks to automation tools (CI, CD)	Need for staff proficient in handling microservices' complexity, or re-train them to do so.[Automation!]

Reduced complexity for maintenance and upgradability, making deployment, testing and developing easier	Transaction consistency issues because of distributed databases
Horizontal **scaling** made possible for critical parts	Increased overhead due to the increased amount of requests between microservices
Fault-resilient architecture	Discovery and documentation issues following the increased amount of smaller application parts
Shorter lead time	Wrong “cuts” may have detrimental consequences down along the line
Lightweight communication, based on REST principles	

3. What are “squads” and “tribes” at Spotify?

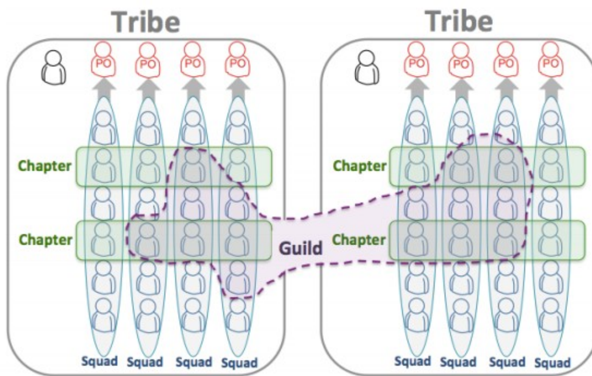


Figure 28: Spotify’s organizational Structure

Spotify is a company that is transforming the music industry by creating a product that can be likened to “a magical music player in which you can instantly find and play every song in the world”.

It has undergone tremendous scaling since its foundation in 2008, both on the software side and on the human resources’ side, and switched from a fully-AWS-dependent architecture to building its own datacenters.

- ***Squads:*** A squad is the basic unit of development at Spotify. It is somehow similar to a Scrum team, and it is designed to feel like a mini-startup. It is a well-rounded cross-functional team, as it has all the skills and tools needed to design, develop, test and release to production. They self-organize and decide their own way of working. Each squad has a *long-term mission* (EX: create the Spotify radio experience).
  Squads are encouraged to apply *Learn Startup principles* such as *MVP* (Minimum Viable product) and validated learning through data collection. They make use of metrics and Alpha/Beta testing to find out what works and what doesn’t work. The Spotify’s motto is “Think it, build it, ship it, tweak it”.
  To promote learning and innovation, each squad is encouraged to spend roughly 10% of their time on “hack days”. In a squad, there is no formally appointed squad leader, but there is a *Product Owner* (PO), who is responsible for prioritizing the work to be done by the team. The product owners of different squads collaborate with each other to maintain a high-level roadmap that shows where Spotify as a whole is heading, and each product owner is responsible for maintaining a matching product backlog for their squad. Each squad has a long-term goal that they pursue over different sprints.
  The squad also has access to an *Agile Coach*, who helps in evolving and improving their way of working by planning 1-on-1 coaching, sprint planning meetings, retrospectives, etc...

Each squad is fully autonomous and has direct contact with its stakeholders. There are no blocking dependencies to other squads.

- **Product owner** - The squad has a dedicated product owner that prioritizes the work and takes both business value and tech aspects into consideration.
- **Agile coach** - The squad has an agile coach that helps them identify impediments and coaches them to continuously improve their process.
- **Influencing work** - Each squad member can influence his/her work, be an active part in planning and choose which tasks to work on. Every squad member can spend 10% of his/her time on hack days.
- **Easy to release** - The squad can (and does!) get stuff live with minimal hassle and sync.
- **Process that fits the team** - The squad feels ownership of their process and continuously improves it.
- **Mission** - The squad has a mission that everyone knows and cares about, and stories on the backlog are related to the mission.
- **Organizational support -** The squad knows where to turn to for problem solving support, for technical issues as well as "soft" issues.

Figure 29: On the left, the organization of a Squad at Spofiy. On the right, the main characteristics of a Squad

- ***Tribes*****:** A tribe is a *collection of squads* that work in related areas – such as the music player, or backend infrastructure. The tribe can be seen as the "incubator" for the squad mini-startups, and have a fair degree of freedom and autonomy. Each tribe has a tribe leader who is responsible for providing the best possible habitat for the squads within that tribe. The squads of a single tribe are all physically in the same office, normally right next to each other, and the lounge areas nearby promote collaboration between the squads. Because of the "Dumbar Number" (i.e.: people cannot maintain stable social relationships with more than 100 people at the same time) tribes are designed to have fewer than 100 people, and consequently reduce the quantity of bureaucracy being passed around in a flat-hierarchy organizational structure.
  Tribes hold *gatherings* on a regular basis, an informal get-together where they show the rest of the tribe what they are working on, what they have delivered and what others can learn from what they are currently doing. This includes live demos of working software, new tools and techniques, cool hack-day projects, etc.

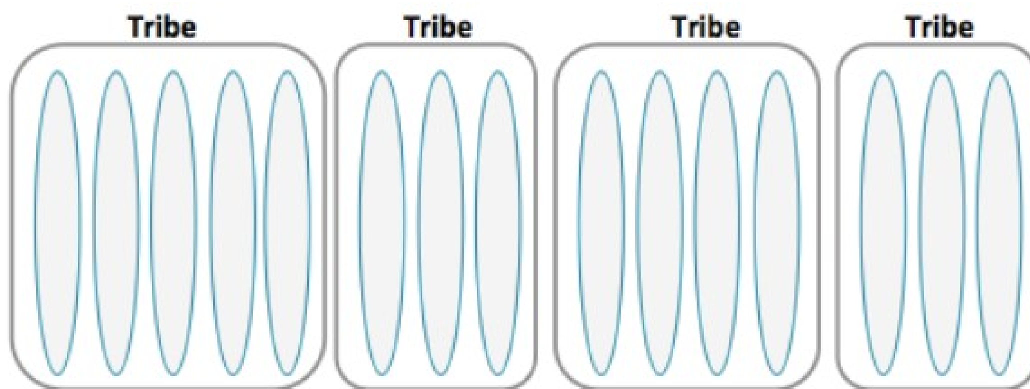


Figure 30: Set of Tribes at Spofiy, each one including many Squads (the ovals within the rectangles)

- ***Chapter***: It is a small family of people having similar skills and working within the same general competency area within the same tribe. This is the glue keeping the company together. Each chapter *meets regularly* to discuss their area of expertise and their specific challenges - for example the testing chapter, the web developer chapter or the backend chapter.

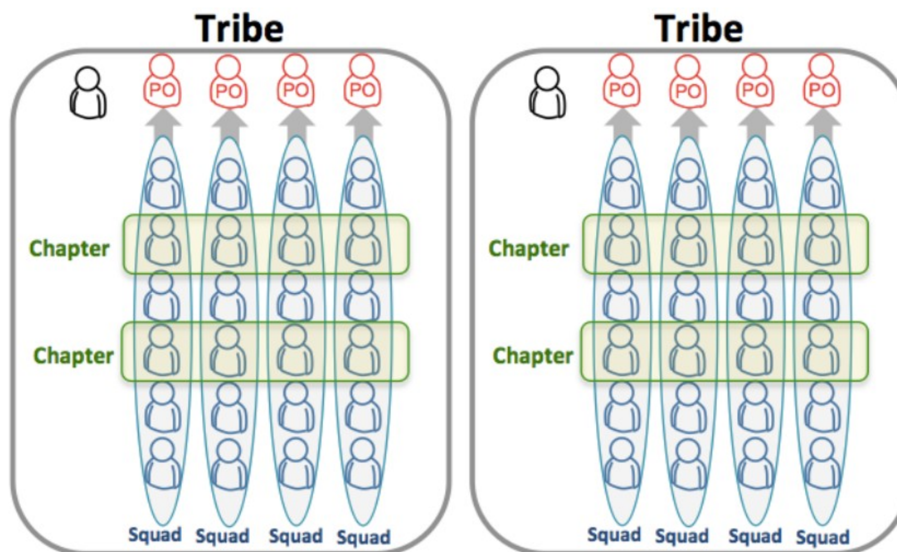


Figure 31: Chapters within Tribes

- ***Guild*: A Guild is a more organic and wide-reaching "community of interest", a group of people that want to share knowledge, tools, code, and practices. Chapters are always local to a Tribe, while a guild usually cuts across the whole organization, and can span across multiple Tribes. Some examples are: the web technology guild, the tester guild, the agile coach guild, etc.
  A guild often includes *all the chapters* working in that area and their members, for example the testing guild includes all the testers in all testing chapters, but anybody who is interested can join any guild.

4. What is Fault-Injection Testing?

Applications need to be designed to tolerate the failure of any service making them up or onto which they depend. Fault-Injection testing consists of intentionally injecting faults into the system to test the system's survivability to faults. This can occur via:

- **Compile-time Injections**: Fault injection technique where the source code is modified injecting a fault in it to see how the system reacts to this fault.
- **Run-time Injections**: A fault is injected into the system at runtime (Ex: making a component stop working and see whether the system or a component still survives this fault).

One example of Fault-Injection testing is the "*Netflix's Chaos Money*", dating back to 2011. The **Chaos Money** is a tool which randomly terminates Amazon AWS VM instances that run inside the *production* environment of Netflix to test the system survivability to failures. The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances. Running such tests in a well-monitored environment can teach a great deal about the weaknesses of the system. Following the success of the chaos monkey, different other ***Simians*** were created, namely:

- **Latency Monkey**: It induces artificial delays in the REST client-server communication layer to simulate service degradation, and measures if upstream services respond adequately. Via very large delays, we can simulate a node or service being down without actually shutting down a service or an instance (onto which other services may depend on)

- **Conformity Monkey:** Finds instances not adhering to best practices (EX: Auto-scaling not enabled) and shuts them down to give the service owner the possibility to start them with proper settings.

- **Doctor Monkey**: Performs a checkup of instances, detecting and removing instances characterized by "unhealthy" conditions (i.e. peaking CPU loads for a large amount of time).

- **Janitor Money:** Ensures the cloud environment is running free of wasted resources. If useless instances are found, these are terminated and disposed.
- **Chaos Gorilla**: Simulated an outage of an entire Amazon availability zone (for example, all the instances of the US West Coast AWS zone).

Such monkeys help ensuring the Netflix architecture can survive (almost) any type of situation or catastrophe that may be taking place.

5. How can "Design for failure" be actually implemented?

The main thing we would like to avoid when designing a microservice-based application is a cascade of failures resulting from tight dependencies of microservices onto other microservices. Namely, when one microservice is down, we would like microservices depending on it to fail "gracefully", responding with a degraded service or a "Service unavailable" message instead of the application crashing.

The approach to be followed when designing a fault-resilient architecture is a *pessimistic one*, assuming every single microservice is bound to fail, eventually. Some principles for coping with every service's failure in an application are the following ones:

- **Redundancy:** Each service should be deployed across redundant cloud components (EX: in different Amazon AWS regions or in different cloud instances)
- **Latency & Fault Tolerance:** Each service should be partition-tolerant, so as to be able to survive network latency that might be occurring in one very network partition without freezing or crashing.

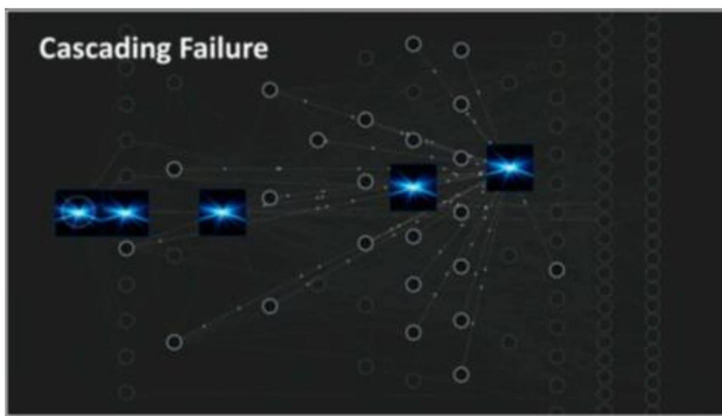


Figure 32: Cascade of failures resulting from one single failure in the Netflix architecture

In a microservice-based architecture built on Cloud, any service call could always fail, because of unavailability of the underlying cloud infrastructure. (EX: Netflix going down on Christmas Eve 2012 because it was fully built on the Amazon AWS East Cost area). A microservice-based architecture is particularly susceptible to failures, as the overall architecture's availability is given by the product of the availability of the different services composing it that have a direct dependency, and a synchronous call.

$$A=\prod_{m\in M} A_m$$

Design for failure

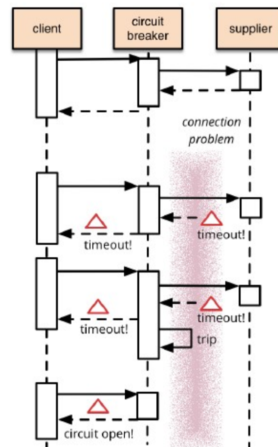


Figure 33: Design for Failure architecture, introducing Circuit Breakers between synchronous calls

Existing applications can be refactored to become fault-resilient exploiting a microservice-based approach based on the following refactoring ways:

- **DB shared by multiple services:**

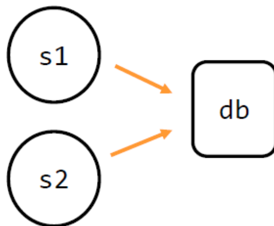


Illustration 1: Architecture of an application having two services (s1 and s2) sharing a DB

Two different refactoring types are possible:

1. **DB Split:** The **DB** is **split** among the two different services, each one getting the part it reads from and/or writes to. However, this is not always possible or easy to do. Also, modifications need to be made to both s1 and s2 and this could be rather demanding in terms of effort.

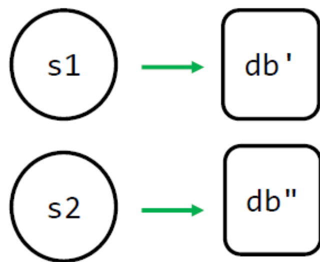


Illustration 2: Refactoring of an application's DB into two separate DBs

2. **DB Manager introduction:** A few changes need to be made to s1 and s2, however a new direct dependency between S1, S2 and the DB manager is introduced.

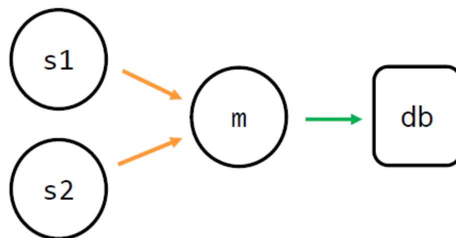
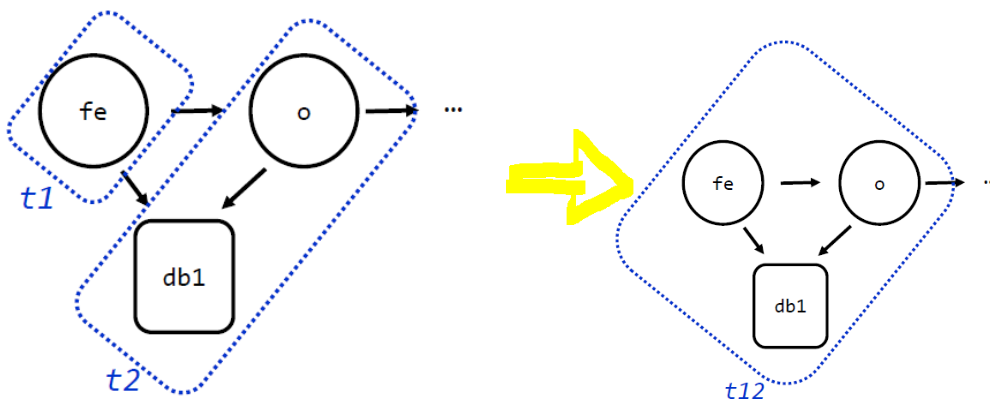


Illustration 3: Insertion of a DB Manager (m) between the two services and the DB

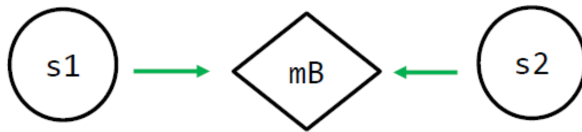
3. Acceptance

- **DB shared by multiple teams**: Two different services managed by two different teams share a DB. A possible refactoring consists of an *organizational refactoring*, where the responsibilities of the different teams are changed: team 1 or team 2 will become responsible for the services based on that DB, whereas the remaining team will be reassigned to a different service.

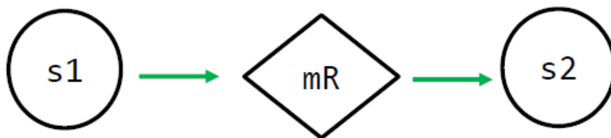


- **Direct Dependency**: One service calls another service via a direct dependency. This is exactly what we would like to avoid to achieve horizontal scaling, as it would lead to a tight coupling between service s1 and service s2.

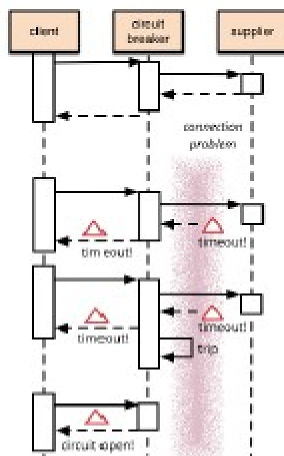
1. **Message broker introduction** : We introduce a message broker, which returns an error to the calling customer in case the service on which s1 depends is not available.



2. **Message router introduction**: It is responsible for delivering a message from s1 to s2 or other microservices.



3. **Circuit breaker introduction:** In case the dependency is not satisfied, failed to response or failed to respond on time, the circuit breaker is responsible for returning an error to the end customer and informing him about the service's unavailability.



4. **Acceptance**

Finally, we can map a microservice-based architecture onto orchestrated containers via Kubernitics, managing different Docker instances.

6.What is Git/Github?

Git is an Open-Source VCS – distributed Version Control System used to keep track of different versions of code and share code among people, both internally to a team and externally (Ex: with releases).

It is based on ***repositories***, which are codebases created to host code for a specific project to which different people can contribute to. Contributions and code changes occur via diff ***commits***, namely a local change in the code (ex: new file created / file edited). After performing a commit, a ***push*** actually pushes the locally changed and committed code to the remote repository. With a ***pull***, on the other hand, one can apply the remote changes made to the local copy of the repository.

A ***fork*** of a repository represents a copy of a remote repository, where changes can be made without affecting the original repository. Eventually, changes from a forked repository can be merged into the original repository. To get a local copy of a repository on one's own machine, a ***clone*** is to be performed.

A repository can have multiple ***branches***, representing different versions and / or features of the codebase (ex: release, development), where code is pushed and tested before being accepted in the master branch (the main one). A branch can be joined with another one by means of a ***pull request.*** A fork is performed automatically when creating a new branch at a fork point. When joining a branch with another branch, that represents a merge (or join) point.

Github is also based on an **eventual consistency** model, where the developer himself is responsible for determining the consistency of the application. A striking example is: we perform a push, but the repository is ahead of us by some commits and some conflicts are detected. So, we need to resolve conflicts between our local files and the remote files to get back to a consistent state.

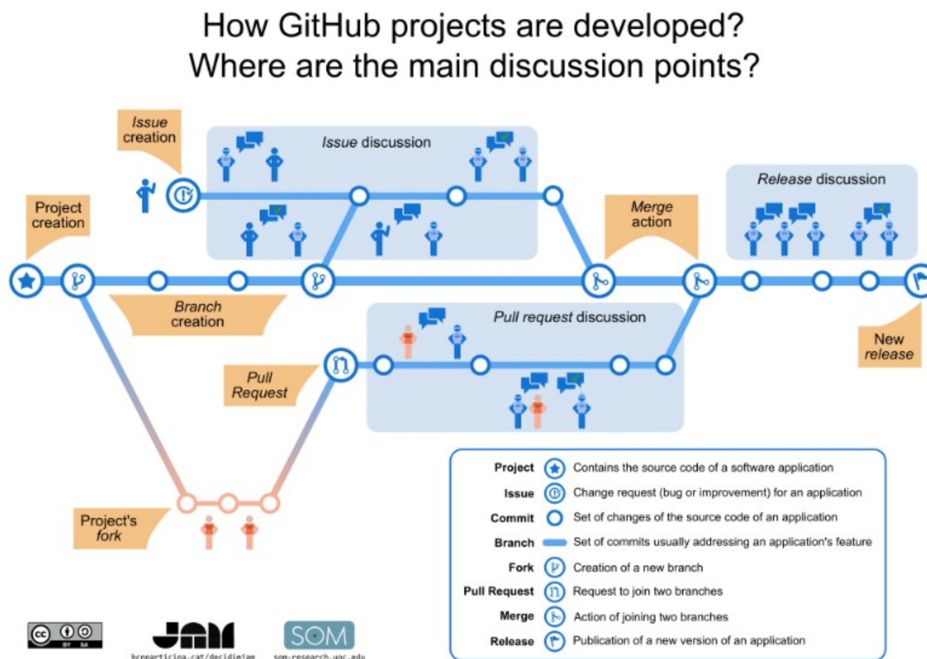


Figure 34: Overview of a Github Project's unwinding and its core components

7. What is Flask?

Flask is a Python-based framework for creating web applications, leveraging the ***WSGI*** - Web Server Gateway Interface toolkit, a simplified interface standard for creating web applications. WSGI runs on standard Web servers (i.e: Apache), however a downside is represented by its *synchronous* nature when performing requests to a service, as the application is blocked till it receives a response from the invoked service.

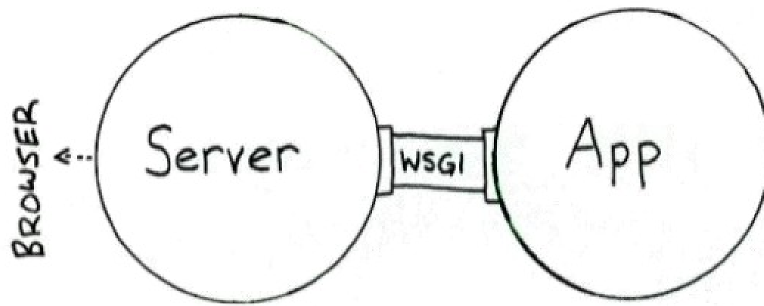


Figure 35: Overview of the WSGI Architecture

Flask was started in 2010 and it is part of the Python ***microframeworks*** ecosystem. A microframework is a set of tools designed to build web apps faster. The difference between a micro-framework and a standard framework lies in the fact in microframeworks that the programmer needs to take as few decisions as possible, hence reliving him from the burden of taking (wrong) decisions that could lead to severe repercussions down along the line.

The Flak microframework is extremely light and by default there are few dependencies to keep track of. On the other hand, should we wish to make use of more advanced functionalities in our Flask application, we would need to specify Flask dependencies manually by listing the set of Python packages our application depends on.

Flask implementation and Flask apps

Flask is based on Python3.2 or higher; the entry point of a Flask class is the flask.app module. Every flask app runs one instance of the Flask class, which is responsible for handling all the incoming HTTP requests, dispatching them to the right piece of code, and returning a response to the caller.

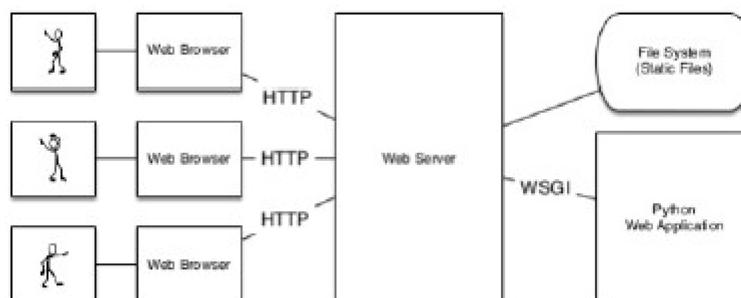


Figure 36: Architecture of a Python-based web application using WSGI

WSGI is a specification that defines the interface between the web server and Python applications. The incoming request is described via a single mapping from Webserver Python application; Python-based frameworks (like flask) will then handle the routing to the right piece of code – And here comes the `“@app.route”` decorator, which specifies the function invoked when a certain part of the web server is accessed and certain parameters passed.

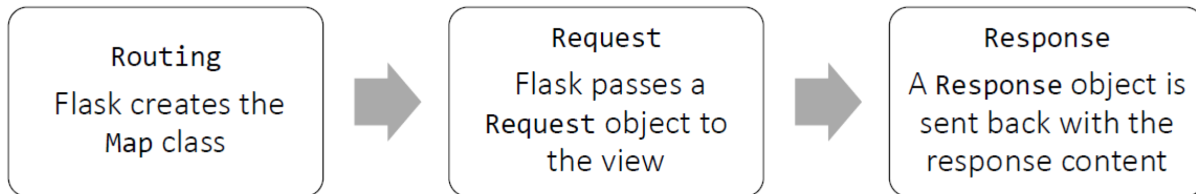


Figure 37: "Under the hood" architecture of Flask

Figure 26 shows how Flask handles incoming requests:

1. A Map class is created to determine if a function decorated with `@app.route` matches the incoming request [Default: Mapper accepts only GET, OPTIONS and HEAD]. Further access methods like DELETE, PUT, POST can be specified "by hand".
2. A request object is created, ensuring each request is treated in isolation with respect to the other requests.
3. Eventually, the response of the method invoking the map class is sent back to the invoker.

8. What is the Model-View-Controller pattern?

The Model-View-Controller design pattern is an architectural pattern commonly used for developing user interfaces, which has also spread to web interfaces lately. The MVC divides the application into three interconnected parts to separate the internal representation of information from the way information is presented to the user. ***Flask*** is also based on the MVC design pattern.

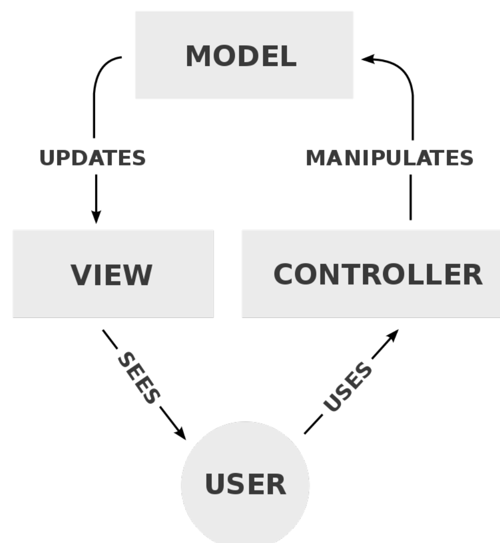


Figure 38: A pictorial Representation of the MVC pattern

The three main components of the MVC design pattern are:

1. ***Model***: Application's dynamic state, which is independent from its representation in the user interface. It stores the data and handles the logic of the application. (Ex: the database containing registered users in the application is one part of the model). It receives data from the controller, and updates the view according to its internal state.

2. ***View***: One output representation of the internal model of the application. Multiple views are possible. (EX: Statistics, home, training objectives). It receives input from the model and outputs the view to the user for final representation.
3. ***Controller****:* Accepts input and converts it to commands for manipulating the model adequately. (EX: the code handling a new user's registration: firstly the user input is validated, then stored in the model).

MVC in Flask:

In Flask, code is organized according to the MVC pattern according to the rule: 1 module = 1 view, where a module is a well-defined functionality, separated from the other functionalities via ***blueprints.*** A ***blueprint*** is a way to organize a group of related views and code underlying the view's funcionality. Instead of registering views and code with the application as a whole, views and code are registered with the blueprint they belong to.

- **Controller:** Controllers, and more specifically controller actions are defined by specifying *routes* in Flask. Concretely, when entering a URL the application attempts to find a matching route and, if one is found, the route's associated controller action is invoked. The *controller* actions represent the set of actions described within a *route function*.

```python
@app.route('/')
def main_page():
    pass
```

Figure 39: Controller action defined in a Route in Flask

- **Model:** After identifying the correct route, the internal *model* is consulted to update the model or fetch necessary data from the database; Then, data is passed to a view for the final representation. In Flask, the model is the internal state of the application (i.e: its DB and any other internal data structures used to store the application's state).

```python
@app.route('/')
def main_page():
    """Searches the database for entries, then displays them."""
    db = get_db()
    cur = db.execute('select * from entries order by id desc')
    entries = cur.fetchall()
    return render_template('index.html', entries=entries)
```

Figure 40: Data being retrieved from the model within a controller action in Flask

- **View:** Finally, we represent the data fetched from the model in the controller action and render it in HTML via Jinja Syntax. This is the final and only representation that will be exposed to the user.

```
{% for entry in entries %}
  <li>
    <h2>{{ entry.title }}</h2>
    <div>{{ entry.text|safe }}</div>
  </li>
{% else %}
  <li><em>No entries yet. Add some!</em></li>
{% endfor %}
```

Figure 41: Jynja Syntax used to represent the model of the MVC to the final user.

4. What is Celery?

Celery is an asynchrous *task/job queue system* based on distributed message passing. Tasks received in queue can then execute asynchronously in the background or synchronously.

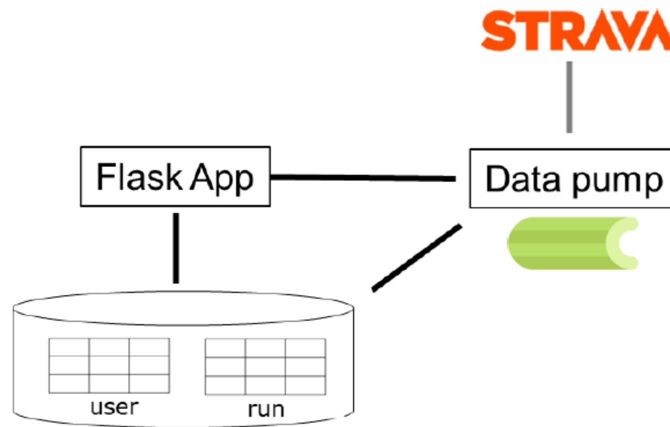


Figure 42: Architecture of the monolith-version of the Strava extension application based on Flask and Celery

In the monolithic version of the application, we used a Celery worker as a message queue to handle the message passing between the Strava application and our Flask-based application. The celery worker got data via *redis* and asynchronously synchronized our application with the data obtained, hence acting as a "data pump" for our purposes. Execution units are called *tasks*, and these can execute either sequentially one after the other or concurrently on multiple workers by means of *multiprocessing.*

Redis: Redis is an in-memory data structure storage, used as a database, cache and message broker.

In the microservice-based version, instead, the Celery worker lies in its own Dataservice microservice; this *dataservice* microservice wraps the DB and exposes a RESTful API, providing only the information seeked by clients. The celery worker still acts as a message-passing queue "under the hood" of a distinct microservice.

5. What is OpenAPI 2.0 / How can you use OpenAPI 2.0?

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of a service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic just by consulting the API.

It is used to define the operations available and the amount of parameters supported by the operations, as well as the parameter types supported by operations supported by the API.

You can use OpenAPI 2.0 by creating APIs on the `swagger.editor.io` website or in a text editor, abiding to the syntax rules of OpenAPI 2.0. We used an interface created with OpenAPI 2.0 as a "backbone" for our application: this interface was able to check the parameter types and the number of parameters passed, raising errors when the wrong parameter types or the wrong amount of parameters was passed. As it is an interface, it allowed us to implement the methods defined in the interface by specifying the actual code.

Software Testing

1. What is Development/release/user testing?

Testing is divided in two main categories, with two main purposes:

1. Show that a program does what it is supposed to do (**validation testing)**
2. Discover program defects before the software is put into production (**defect testing)**

“Testing can only show the presence of errors, not the absence of them”.

- Djikstra

Development Testing

Development testing encompasses all types of testing done by developers when developing the system. The tester of the system is hence the programmer herself who developed the software. For more critical systems, a separate group within the development team may be involved in the testing of the system. There are three main stages of development testing:

1. **Unit Testing**: It is the process of testing single program components, such as methods or object classes. When testing *methods*, one should call methods with different possible input parameters according to Partition testing of Question (2). When testing *object classes*, one should design tests to cover all the features of the object. This implies testing the setters and getters and getting of all attributes in the object, and test the object in all possible states: this would imply simulating events that would lead to the object to be in all possible states (however, this is infeasible, in practice as the amount of states is exponential). Generally, test methods should be called in isolation in unit testing, but when this is not possible a sequence of actions may be necessary.
   Whenever possible, Unit testing should be *automatized*, leveraging existing Unit testing frameworks such as Junit or PyTest. Such testing frameworks provide a way for running tests in a simple manner within a a few seconds, and a GUI for showing the success or failure of the tests run. Because an object class being tested or a method often has some kind of dependencies on other objects that need to be setup before using the object or calling the method, an automated test has three parts:
   1. ***Setup part***: The system is initialized with the dependencies required by the system part being tested; the input and the expected outputs are set.
   2. ***Call part***: The method or the object being tested is called.
   3. ***Assertion part***: The actual output of the call is compared with the expected output. If the assertion evaluated to true, the test has been successful; if false, the test failed.

   Testing external objects may lead to a large performance burden when testing, so we may wish to resort to ***mock objects***, which simulate functionalities of an external dependency of our piece of software being tested. One such example is an access to a remote database, which could be significantly expensive in terms of access time and in complexity in its setup. Using mock objects reduces the complexity in the test setup and reduces the access time.

   In our BeepBeep app, we made use of mock objects when testing functionalities exposed by the Strava API by simulating the presence of runs and users returned by the Strava API.

   Effectiveness of Unit Testing:

   Because testing is an expensive and time-consuming operation, it is important to choose effective unit test cases. Effectiveness has a two-fold meaning, analogous to the general meaning of testing:

   1. **Unit Validity:** The test case should show that, when used as expected, the component tested does what it is supposed to do.

2. **Unit Defects:** The test case should also show that, if there are defects in the component, these should be revealed by test cases. (if a unit test does not pass, we would identify the unit not working straight away).

Therefore, two different types of unit test cases should be designed: the first type showing that the component operates correctly when given a valid input, and the second type showing that abnormal inputs do not crash the component, but such inputs are still processed by the system.

Test cases can be chosen based on:

- **Partition Testing**
- **Guideline-based testing**

2. Component Testing: Software components are often made up of several interacting objects "under the hood". The functionality of a component, made up of different objects, is accessed via the interface it exposes. Testing of a component therefore implies verifying that the component interface behaves according to its specification. One can assume that the individual objects encapsulated by the component have been already validated via unit testing.

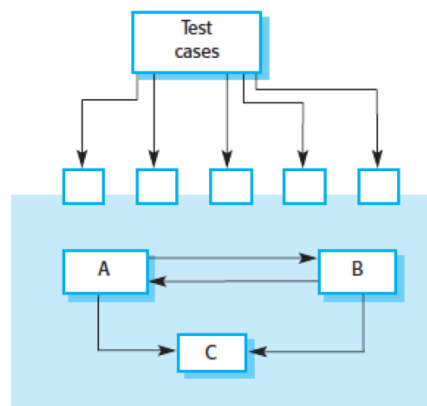


Figure 43: Test cases testing an interface of a component encapsulating different objects interacting with one another

One large component may be made up of other components. Such component is known as a *Composite Component*. Testing such composite component requires testing different components interacting with one another, as interface errors in the composite component may not detectable by testing the individual components only. Different types of interfaces exist between program components and, consequently, different types of interface errors may occur.

1. **Parameter Interfaces**: Data or function references are passed from one component to another one (e.g: in object methods).
2. **Procedural Interfaces**: One component encapsulates a set of procedures that can be called by other components. (e.g.: in objects and reusable components)
3. **Message passing interfaces**: One component requests a service to another component by passing a message to it. This occurs in a request-response way, and is typical of object-oriented or client-server systems
4. **Shared memory interfaces**: A block of memory is shared between components. Data is placed in memory by one component and is read by a different component (EX: used in embedded systems).

Interface errors are one of the most common forms of errors in complex large systems, and they can be grouped into three main categories:

1. **Interface misuse**: A component calls the interface of another component and passes the wrong type for the parameters, the wrong amount of parameters, or parameters in the wrong order. In some cases, such errors are raised at compile-time already.
2. **Interface misunderstanding**: A component misunderstands the specifications of the interface and hence calls the interface in an erroneous way. (ex: calling a binary search on an unsorted array).

3. **Timing errors**: These errors occur in real-time (embedded) systems that use a shared memory or a message-passing interface. Called and calling component operate at different speed, and the caller accesses out-of-date information, which have not yet been updated on the called component's end.

Testing interfaces is difficult, as some interface faults may only occur under unusual conditions, for example an error may be raised in case multiple components have already had a fault. Some *general guidelines* for testing components are the following ones:

1. Pass parameters to a procedure at the extreme ends of their possible range
2. Always test null pointers when dealing with pointers
3. When a component is called through a procedural interface, design tests that cause the component to fail.
4. Use stress testing in message-passing systems. This may reveal timing problems.
5. In shared-memory systems, vary the order in which components are activated.

NB: Sometimes, static code inspections may be more useful than a lot of dynamic testing does when looking for interface errors (or security vulnerabilities).

3. System Testing

During development, components are integrated to create a version of the system. Testing the new version of the system will check that components interact correctly with one another to deliver the awaited *emergent behavior* of the system. The emergent behavior is the behavior of the system when different components are put together. System testing involves *regression testing* (i.e: checking that a newly implemented component has not broken the functionality of other components), as well as *interaction testing* (i.e: checking that the newly integrated component correctly interacts with other components)

Components may either be single functionalities created by one or a team of developers, or even an off-the-shelf system integrated with other off-the-shelf systems.

One way to check the emergent behavior of the system as a whole is by ***use case-based testing.*** We identify a set of use cases for the system, then we test the use case, which will involve testing many different components. This approach is somehow similar to scenario-based testing, as we create user sequence diagrams depicting the user's requests and the components interacting with one another to satisfy such requests.

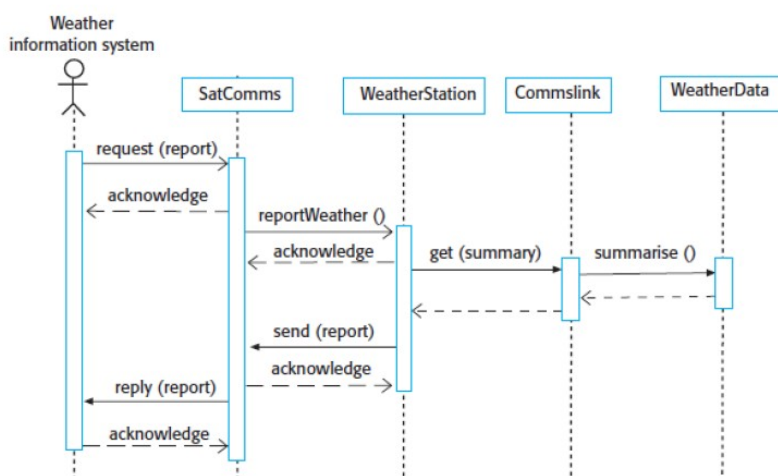


Figure 44: User asking for a Weather report to a Weather report system, shown in a User sequence diagram

Testing all possible sequences of interactions (system action paths) may be impossible in large systems, so testing usually occurs according to some company-based or experience-based policies.

EX:

- All program statements should be executed at least once
- All functions available from the menu should be tested at least once
- Where user input is provided, all functions must be tested

Release Testing:

Release testing consists of testing the whole system before it goes into production, with the purpose of checking that a particular release of a system is fit for usage outside of the development team, and it does not fail when used by normal system users. The *primary goal* of release testing is to convince the system supplier that the system does meet its requirements wrt. functionality (validation testing), performance and dependability and it is good enough for usage by final system customers.

Generally, release testing is performed by an external testing team in a *black-box* fashion, where tests are just derived from the functional specifications drawn up-front. Release testing can be thought of as a form of system testing.

Different approaches to release testing exist:

1. **Requirements-based testing:** As a rule of thumb, it is a good practice to make requirements testable. This means that requirements should be written so that one or more tests can be written to verify whether that requirement has been satisfied. In requirements-based testing, a systematic approach can be applied, where test cases are derived directly from the requirements. Once the test cases pass, we can be reassured about the functioning of the system.
   In practice, you usually have to write several test cases to cover the different situations implicitly present in a requirement. It is also good practice to keep some form of "traceability" between tests and the requirements they test.
2. **Scenario testing:** It is an approach to release testing where you devise typical usage scenarios for the system and create tests according to such usage scenarios. A scenario is a narrative story that describes one way in which the system may be used. Scenarios are analogous to *Usage stories*: they are defined in the requirements engineering process, and may be re-used as testing scenarios. A well-designed scenario should be realistic and should be sufficiently relatable by the system stakeholders that they will be convinced about the importance of the corresponding test to pass. Also, in a scenario-based approach, you test several different individual requirements, as well as the combination of different requirements.
3. **Performance (load) testing**: After the different requirements have been fully integrated into the system, the system is tested for its emergent non-functional properties, such as *performance* and *reliability*. Performance tests ensure that the system is indeed able to process the intended load under normal conditions.
   - To *validate* the performance of a system, namely to ensure that the system's performance meets its intended *performance requirements*, one may have to construct an ***operational profile***. An operational profile is a set of tests that reflect the actual mix of inputs that will be handled by the system, weighted by the probability of the inputs to occur.
   - For *defect* testing in performance testing, experience has shown that defects are often discovered around the system's limits. This can be tested via ***stress testing.*** By this approach, we test the system by increasing the load until the system's performance degrades to a point where the performance is no longer acceptable. When reaching the "breaking point" of the system, we check that the system "fails soft" and does not cause heavy data corruption or loss when under heavy load.
     Stress testing also helps in revealing defects that only show up when the system is fully loaded, and may not occur during normal usage. Some defects not causing important full system failure during light-load operation may instead lead to system failure when the system is heavily loaded.

User Testing:

User or customer testing is a stage in the testing process where final users or customers test the system, providing comments and advice about the state of the system. This may be a *formal test* with the supplier or may involve more *informal testing* to check that the system does what it's supposed to do.

User testing is **extremely important** because it is practically impossible for a developer to replicate the *environment* where the system is going to be used in. Also, developers just base their testing on the requirements they have at hand, but these may not include other factors that affect the actual usage of the system. (EX: Consider a piece of software being used in a hospital with patients coming in and doctors or nurses going around: in no way could a developer replicate such scenario unless he were to go and actually test the software on site). There exist three different types of user testing:

1. **Alpha testing**: The system is made available to a restricted group of users, who work closely with the development team, testing *early releases* of the software in their environment, reporting possible defects in the system and ensuring the system behaves as expected. Alpha testing is often used with software products or apps.
2. **Beta testing:** The system is made available to a larger group of users, who test *later releases*, possibly more stable than the ones made available to the alpha testing users. Again, users report issues to developers directly. Beta testers may be early adopters of the system or the software may be made publicly available for anyone willing to experiment with it, in order to report possible issues arising during the interaction between the software and characteristics of the operational environment the software is used in.
3. **Acceptance testing**: Customers test a system to decide whether the acceptance criteria (eventually specified in the contract) are satisfied, and the system is ready to be finally deployed in the customer environment. The ***Acceptance testing process*** involves multiple phases: firstly, acceptance criteria are *defined* in the system contract and approved both by developers and the customer. Then, acceptance testing is *planned*. The resources, time and budget for acceptance testing are decided. Acceptance tests are then derived, which should cover both functional and non-functional characteristics of the system and ought to fully cover the acceptance criteria. After running all the acceptance tests of the final system (possibly in the customer environment), the test results may be negotiated and a partially-working version of the system may be deployed, while waiting for the fully-fledged one to be fixed. If the system is rejected, then further fixes need to be made and the program's deployment is delayed to a more stable state.

   In agile development and XP programming , there may not be a separate acceptance testing activity. Testing is done by developers themselves and is embedded in the user stories provided by customers. Such user stories and their corresponding tests are equivalent to acceptance tests.

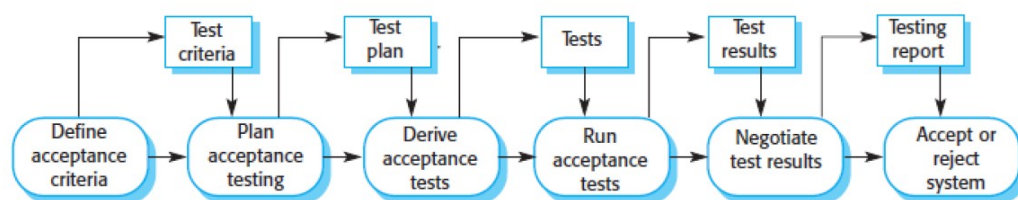


Figure 45: Overview of the Acceptance Testing Process

2. What is Partition Testing?

Partition testing is a technique for creating test cases for both systems and system components, which requires partitioning the inputs into sets of functionally-equivalent inputs and functionally-equivalent

outputs. Such partitions are called *Equivalence Partitions*, as the elements in each partition are treated by the system or by the system components in an analogous way.

Input data and output results can be thought of as members of sets with some common characteristics. For example, the sets of positive and negative numbers. Both input elements and output elements can be grouped respectively into *input equivalence partitions* and *output equivalence partitions*.

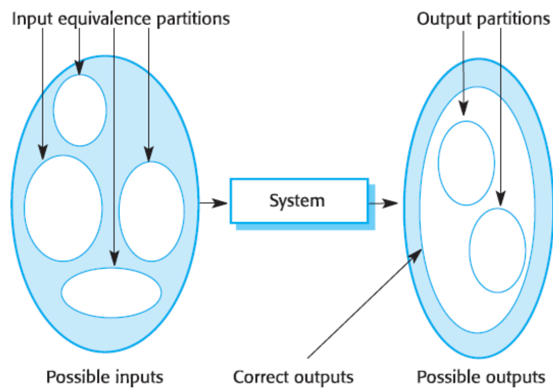


Figure 46: Inputs split into input equivalence partitions and output elements split into output equivalence partitions

Partitions can be identified through the program specifications, user documentation or by experience (**black box testing)**, to check where the program could possibly produce erroneous values

After identifying a set of partitions, you choose a set of test cases from the middle of these partitions (validation testing) and on the boundary of these partitions (defect testing) to ensure all the equivalence partitions are covered by test cases.

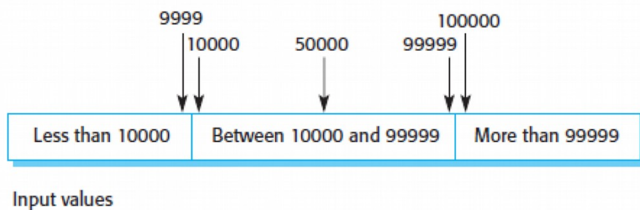


Figure 47: Input partitioning example

Alternatively, test cases can be derived according to the following guidelines:

Testing guidelines

- For sequences
  - Test with sequences which have only a single value
  - Use sequences of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the sequence are accessed
  - Test with sequences of zero length
- In general
  - Choose inputs that force the system to generate all error messages
  - Design inputs that cause input buffers to overflow
  - Repeat same input (series) many times
  - Force invalid outputs to be generated
  - Force computation results to be too large or too small

Figure 48: Testing guidelines for picking test cases

3. What is Test-Driven Development (TDD)?

Test-Driven Development (TDD) is an approach introduced with XP programming to program development, which consists of writing tests before writing the code. After a test is laid out – generally requiring a few lines of code to be passed – just enough code for the test to be passed is written. Passing a test is the fundamental part of the development – as long as the test is not passed, no further features ought to be implemented.

Via this approach, code is developed incrementally in small chunks. The main benefit of Test-Driven Development is that it helps programmers clarify their ideas about what a piece of code it supposed to be doing. To write a test, you need to understand what functionality is intended to be tested This understanding makes it also easier to write the corresponding code to pass the test.

An automated testing framework (e.g: Junit) is fundamental for TDD, so as to automatize regression testing.

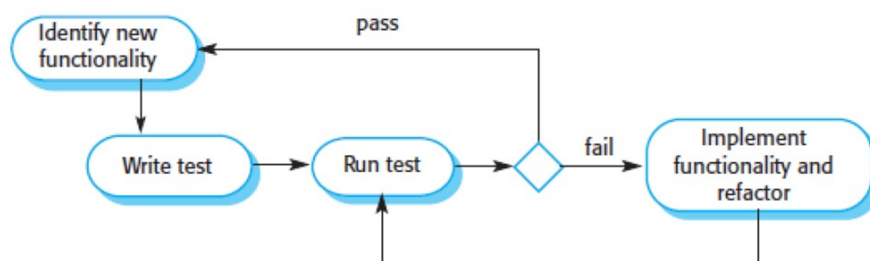


Figure 49: Approach to Test-Driven Development

Figure 49 shows the four-phase approach followed in TDD to implement new functionalities:

1. Identify the new functionality you want to implement, usually requiring little code to be implemented
2. Write a test for that functionality and implement it as automated test
3. Run the test, which will surely fail because the functionality has not been implemented yet, along with all other tests for preceding features that should (hopefully) pass.
4. Implement the new functionality and re-run all tests. If they all pass, then move on to next functionalities. If they fail, looking at the failing tests and patch up the code causing issues.

The benefits of TDD are the following ones:

1. *Code Coverage*: Because each code segment has at least one associated test, every piece of code must have been executed at least once. As code is tested incrementally while developing, defects are discovered early, and (usually) do not make it into production.
2. *Regression Testing*: Thanks to the support offered by automatized testing frameworks, it is always possible to check that a new feature has not broken previously implemented features.
3. *Simplified debugging:* Especially when introducing new features, it is very easy to find out where a problem lies by looking at the failing tests – and the corresponding functionalities they test.
4. *System documentation*: The tests themselves are the documentation of the program.

4.What is Load Testing?

See Performance Testing under "Release Testing" of Question 1. As part of one lab on Load Testing, we experimented with the "Boom" and "Molotov" tools for stress-testing a web application.

User Stories

1. **What is a User Story (for) ?**

A user story is a high-level artifact representing a high-level small usage requirement for the system to build, which delivers value to the customer. User stories are much smaller than use cases or usage scenarios laid out in scenario testing. A good story should be implementable in 1-2 working days and must not take longer than one sprint's duration. Long stories are split into shorter stories.

A user story represents a system functionality desired by the user and described by the user himself from his own point of view. Functionalities are implemented from the user's point of view by the JIT – Just in Time programming methodology, focusing on implementing "just enough" of the underlying levels' dependencies upon which the high-level user story depends on via ***vertical slicing***. (EX: If you need to implement a payment system, you would implement "just enough" of the DB, backend and frontend necessary to get the system up and running with that functionality).

User stories allow to visualize the amount of work left to be done, as it happens with a good Kanban board, and also allow to distinguish more valuable work from least valuable work via *prioritization.*

User stories are created within different phases:

- **During JIT planning / meetings with stakeholders**: During a conversation with stakeholders (customers), you will discuss the details of the user stories you will be implementing as part of the project's committments.
- **During iteration planning:** When we plan our sprint, we may create a set of user stories that will be worked on in the upcoming iteration.
- **During implementation**:If realising one user stories is too large, then we may have to split it into multiple user stories.

Vertical Slices
over
Horizontal Slices

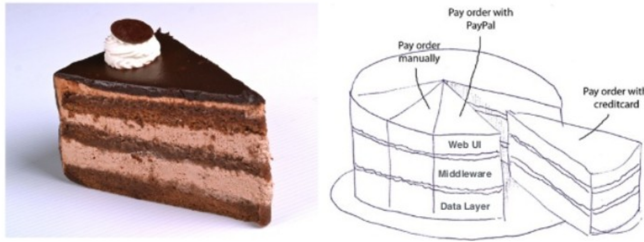


Figure 50: Implementation of a user story as a vertical slice of functionalities spanning different layers

Characteristics of user stories:

1. User stories are written by stakeholders
2. Use the simplest tools to manage user stories
3. User stories aid in remembering non-functional requirements.
4. User stories carry the *estimated effort* required to implement them. The effort is decided by the developer needing to implement the user story.
5. User stories carry a *priority*, which is set by the project stakeholder (in Spotify, Product Owner). User stories are sorted according to the priority they carry.
6. [Optionally] Carry a unique Identifier to link user stories to other artifacts, for example acceptance tests.

There exist different ways of designing user stories, but the most widely adopted approach is based on the following format:

As a *<role>*
I want to *<requirement>*
So that *<value>*

As a
I want to

So that

Figure 51: Template for a user story

2. What is the priority of a user story?

The priority of a user story is the importance of the user story to the customer. User stories are usually organized in a stack, with high-priority user stories lying at the top of the stack for developers to implement them. Low-priority user stories lie at the bottom of the stack and will be the last ones needing to be implemented. The priority is set by the project stakeholder.

User stories are sorted according to the priority in decreasing order; priority is either expressed via a numerical range, usually from 1 to 10 (with a lower number representing the top priority as in Teletraffic Engineering), or via ordinal attributes: low, medium, high. The granularity depends on the amount of stories that need to be implemented and their importance. (e.g: we don’t want to end up having very very very high importance for stories, but we’d prefer to keep it as simple as possible).

N.B: Defects found in the system that need to be fixed are prioritized too, like user stories, based on the criticality of the issue.

3. What is the effort/size of a user story?

The effort/size of a user story is the time required to implement the user story in practice. It is set by the developers needing to implement the user story. One way to estimate the size of a user story is by assigning user story points to each card, where each story point represents a certain implementational time effort.

User stories should neither be too fine-grained to require less than one hour to be implemented nor too coarse-grained to require more than one sprint to be completed. Even if developers may feel they do not have the right estimating skills at first, they usually hone their estimation skills quickly enough when they are forced to live up to the estimates they make.

The effort of a user story in the XP framework is estimated for a pair of programmers working in a pair-programming way.

4. What is an Epic?

An *Epic* is a large user story that cannot be delivered in a single iteration, and is broken up into multiple smaller user stories implementable over multiple iterations. Epics are typically lower-priority user stories because, as the epic makes its way to the top of the user stories' stack, it is re-organized into smaller higher-priority user stories, which are easier to manage and implement.

Unless a certain part of the Epic has a higher priority and needs to be addressed quickly, commitment to implement the Epic should be delayed to a later point in time: by implementing the whole Epic from day one, you may end up wasting time on something you may never get to use or poorly designed at the beginning. One suggestion is to defer commitment, in a JIT (Just-In-Time) basis to increase the overall productivity and focus on the most urgent tasks first.

Epics allow to keep track of large loosely defined ideas in the backlog without overpopulating the backlog with multiple items that will never be addressed. The user stories associated with that epic represent the various aspects of the solution you need to deliver, or the options you have for satisfying that need.

Splitting the monolith

1. When and where to start splitting a monolith codebase?
A monolithic codebase grows overtime, as changes are applied to it and features are added over time. As the codebase grows, *cohesiveness* is lost, and *code coupling* is no longer loose, but coupling becomes more and more tight – with messy dependencies .

- *Code Cohesiviness*: It is a measure of the degree to which elements within a module belong together. Initially, code of a monolith is very cohesive, with elements residing close to elements belonging to the same module. Over time, code cohesiveness is lost and every kind of code belonging to different modules starts interacting.
- *Code Coupling*: It is the degree of interdependence between software modules. Ideally, we would like it to be loose (i.e: independent features' code is independent from the code of other features).

As these two code quality metrics are lost, maintenance costs increase.

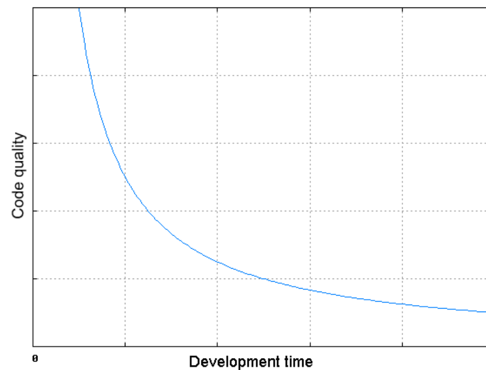


Figure 52: Code Quality over time for a monolithic codebase

It is generally a bad idea to spend effort and time splitting a monolithic application too early in time, when the code has not yet lost all of its cohesiveness and coupling properties. Instead, as a rule of thumb we consider:

"Split the monolith when it becomes a problem(pain in the ...)"

The key idea for splitting a monolith consists of identifying seams that can become service boundaries.

A ***seam*** is a piece of code that can be treated in isolation and worked upon without impacting the whole codebase.

We can identify ***seams*** based on *bounded contexts*, which represent cohesive, yet loosely coupled boundaries in an organization. Bounded contexts can be thought of as a cell, delimited by its own *membrane*. What is inside the bounded context needs not to be communicated outside, but stays within the cell thanks to the membrane (the boundaries of the context).

Additionally, programming-language specific delimiting concepts can be useful for identifying bounded context (EX: Java's packages, C++'s namespaces).

Approach to identify seams:

1. Firstly, we *identify* the *high-level bounded contexts* that exist in the monolithic codebase and that represent the high-level functionalities of the code (EX: Tools like Structure 101 are useful to visualize dependencies among packages).
2. We create packages reflecting these high-level bounded contexts, and move code into the packages. This can be done via *refactoring* tools in an incremental way, bit by bit.
3. Move code left over to a new bounded context.

Where to split the monolithic codebase:

After identifying the seams and moving the code into bounded contexts, we are ready to split the monolithic codebase into *microservices*. As a rule of thumb, an incremental approach is recommended, to limit the impact of wrong decisions and to learn microservices "on the field", bit by bit.

Which seams do we pick first to restructure the monolithic codebase into microservices?

"Start from where you can get the most benefit"

The "most benefit" can be intended in several different ways:

- *Pace of change:* If we know that there are going to be lots of changes in a particular part of the code, we start splitting from that one part first.

- *Team Structure:* If there are different teams working on different parts of the code, we restructure the code so as to let the teams work on separate and independent parts of the code.
- *Security*: If we split the service handling security, we can potentially improve the security level of the application
- *Technology*: Split a particular seam that could benefit from a new technology into a microservice (EX: One part of the system could benefit from a new ML library available in Python).
- *Tangled Dependencies*: Pull out the seam that has least dependencies on it and turn it into a microservice, as it is the easiest one to pull out.

2. How to split databases? (e.g: how to break foreign key relationships?)

Databases are often used in a monolithic codebase to integrate multiple services. We need to find seams in our databases so we can split the different databases cleanly.

1. *Understand* which parts of the code read from and write to the database (EX: Tools like **Hibernate** can help with this)
2. *Detect* database-level constraints (e.g: database-level foreign-key relations across tables). Also, tables may be shared by different bounded contexts too and may have some foreign-key relationships.

Breaking foreign key relationships

Let’s consider the following example:

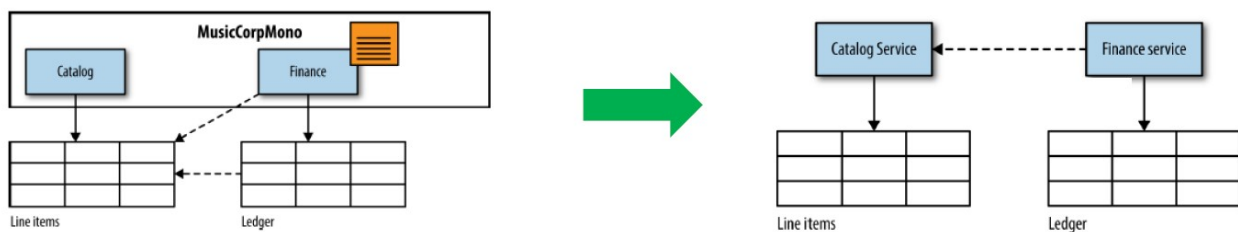


Figure 53: Originally, the Ledger table has a foreign key pointing to the UID of Line Items. Following the refactoring, the foreign key is dropped.

Finance code accesses line item table in case we want to generate reports for various people in the organization, and a foreign key relationship is established from the “Ledger” table to the “Line items” table, *Refactoring:* We get rid of the foreign-key relationship, avoiding the code in finance from reaching the “Line item” table, then we wrap the catalog service via an API to expose the catalog’s services.

The finance service will then be able to access the catalog service by performing calls to its API
+ Decreased dependency among services
- Overhead introduced, as we may need to perform multiple API calls
- Foreign key relationship managed in the service, lost at the DB-level (consistency issues in the DB)

Shared Static Data

Consider the case where multiple services access one database, and yet this database's data is never (or raely) changed. The typical example is with country codes being stored in a database. We may have 3 solutions at hand:

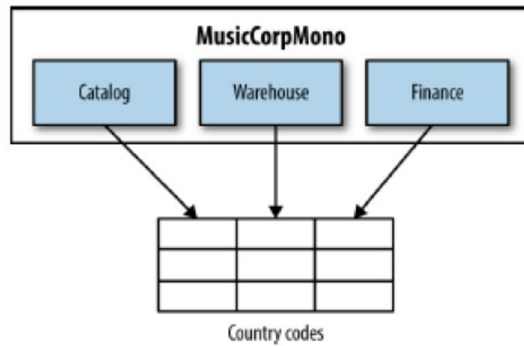


Figure 54: Example of shared static data stored in a database accessed by three different services

1. For each service (and package), *duplicate* the table containing the shared data
   - Consistency issues in case some static data is changed
2. Treat the shared static data as code (EX: put it into a configuration file) and move it from the DB to the services themselves.
   - Some consistency issues, but easier to manage a single file rather than replicating DBs.
3. Create a new service wrapping the static data
   - May be too much of an overkill?

Shared Mutable Data

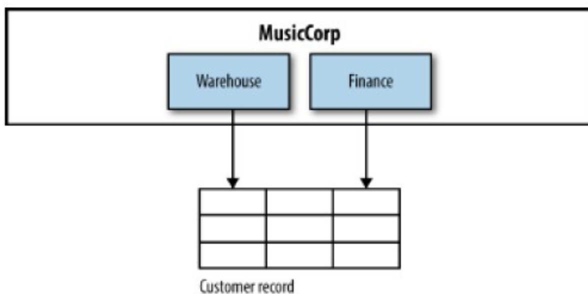


Figure 56: Application having the "Warehouse" and "Finance" services accessing the Customer Record table

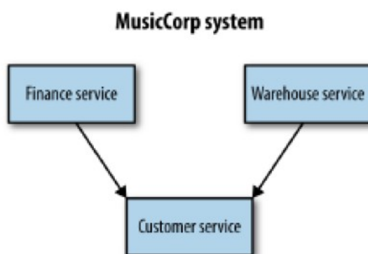


Figure 55: Application refactoring into three different services, where the customer service was refactoring into its own service

Let's consider the case where both the 'warehouse' and 'finance' services access to and write to the "customer record" table, in that the customer is modeled in the database abstractly. In this case, we have a *domain concept* that isn't modeled in the code, but is actually present in the database.

For a clean refactoring, we make the Customer a concrete entity by moving it into its own package and finally creating a service for it; we also make it expose an API which will be accessed by the 'Finance' and 'Warehouse' services.

Shared Tables

Let's consider a table storing data accessed by two different services (EX: *Catalog*: store name and price of the record we sell, *Warehouse* keeps an electronic record of inventory). Both these pieces of information are kept in the same item table!

Our refactoring consists of splitting the table into two separate tables, each one containing data related to separate concepts and being accessed by a different service.

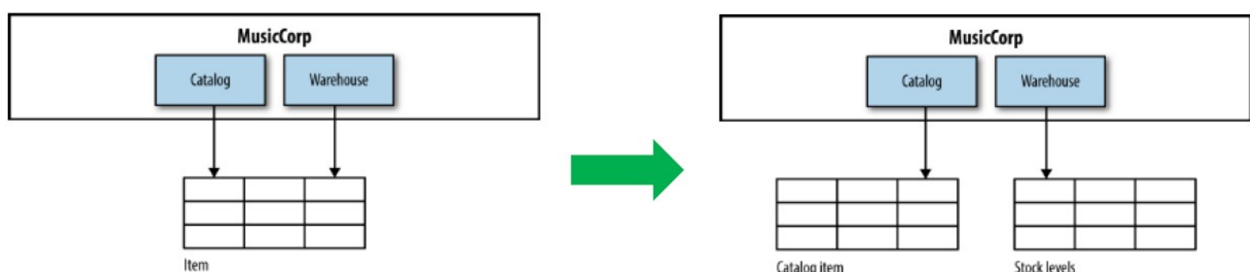


Figure 58: Example of a table having data shared by two services, and its refactoring by splitting the table into separate tables

NB: Before actually splitting the application into microservices, it may be a good idea to proceed in steps instead of performing a big-bang change from a Monolithic Architecture to a Microservices-based one, which would be extremely complex to be changed in one single large step.

1. **Step:** Split the whole schema into multiple schemas, with each one being accessed by a different *bounded context [*Keeping the ability to keep the application code together, and reverting/tweaking changes without impacting any consumer of the services].
2. **Step**: Split the application into microservices based on the split schemas.

➔ This architecture is bound to increase the amount of database calls and the overhead connected to the increased amount of requests performed.

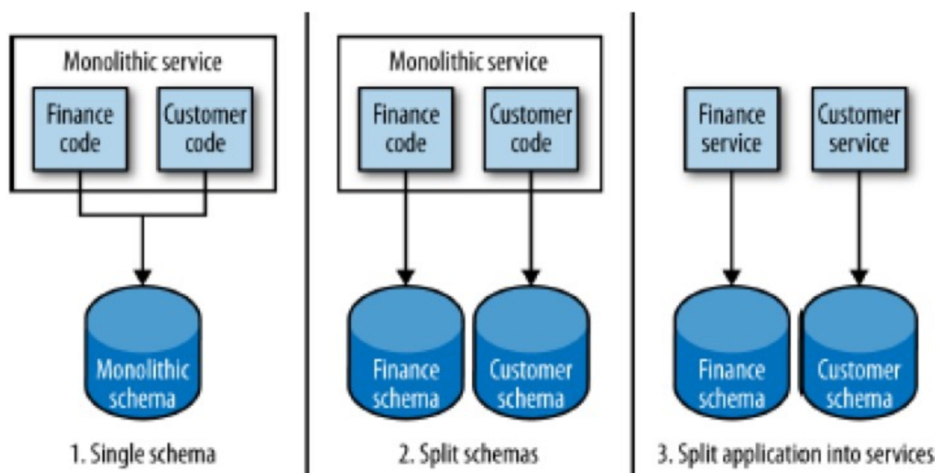


Figure 59: Pictorial representation of the staging of a service separation by first splitting the schemas, and then splitting the application into services.

3.What about Transactions when you split?

Transactions allow us to group together multiple different activities that take our system from one consistent state to another consistent state. Issues with transactions arise when having to ensuring consistency among different (distributed) schemas though. They allow us to say:

All-or-nothing: “*Either all events happen together, or none of them happen*”. If all events are successful, then the transaction does change the application’s state; otherwise, if at least one event fails, the application’s state is rolled back to the last known consistent state, rolling back the events that modified the tables.

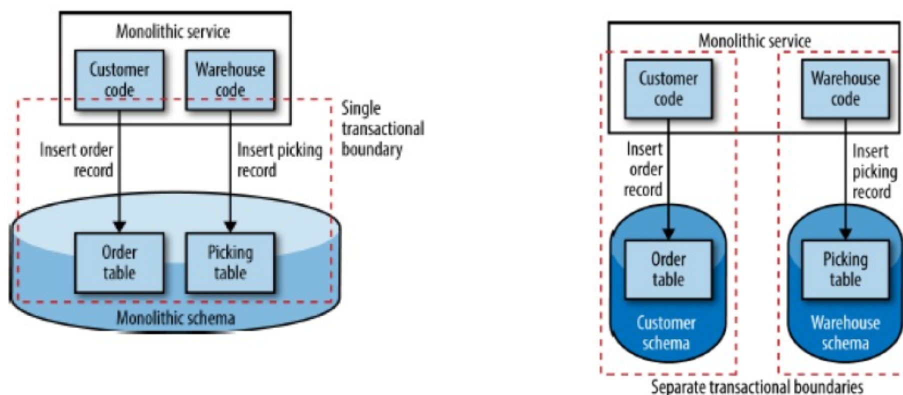


Figure 60: Separate transaction boundaries spanning over multiple schemas

Figure 61: Single transactional boundary within

a monolithic schema containing multiple tables

If we have one single monolithic schema, then we are able to update multiple tables in a single transaction, allowing for the schema to stay in a consistent state before and after the transaction has occurred. This ensures *transactional safety*

However, if we have multiple schemas, then *transactional safety* is lost, as a transaction will be spanning multiple separate transactional boundaries.

(EX; Insert order table fails, but insert into the picking table fails leaves us in an inconsistent state).

Solutions to Transactions over multiple Schemas

1.*Abort the entire operation*: If one single operation within a transaction fails, we unwind the committed transactions by performing *compensating transactions* in order to get the database back again in a consistent state. The question now is:

“What if the compensating transactions also fail?”

- ➔ We would need to either retry all the compensating transactions (potentially forever) or allow some process to clean up the inconsistent state at a later stage. Also, the logic for handling failing transactions may reside in many different parts of the codebase’s services.
  [If we need to retry all the failed transactions for 2,5, 10,… transactions to ensure their consistency, the whole process becomes **unscalable**]

2.*Use Distributed Transactions*: This is an alternative approach to manually orchestrating compensating transactions. Distributed transactions span multiple transactions within them using a “*Transaction Manager*”, which orchestrates the different transactions being done. We hence make use of a “***Two-phase commit***”, which works in two phases:

- *Voting phase*: Each participant (a *cohort*) tells the manager whether its local transaction can go ahead.

- *Decision phase*: If the transaction manager gets a "yes" vote from all participants, it tells them all to go ahead and actually commit their transactions. If it gets at least one "no" vote from one participant, then it rollbacks all the transactions on all parties.

Issues:

- The resource manager represents a single Point of Failure, as the pending transactions would never complete.
- Resources are locked on all participants while waiting for the transaction to complete (leading to scalability issues).
- If one single participant fails to respond during voting, then everything blocks. (each participant is a single point of failure)
- We assume that a commit cannot fail after voting (if a participant says yes, then it WILL commit).

3.*Eventual Consistency – Try Again later*: If one single event fails, then the database is an inconsistent state. We hence report that the operation could not be completed successfully, and invite the user to retry at a later point in time. This is a form of eventual consistency, in that we accept the fact that our database is in an inconsistent state, but we will get to a consistent state at some point in the future, in a best-effort way. No strict transactional boundary is implemented [Netflix and Amazon follow this approach, but banks cannot apply this consistency model and would need to implement some sort of distributed transactions].

4. What is the Saga Pattern?

The SAGA pattern is a pattern that can be used to achieve eventual consistency by managing failing distributed transactions. It can be used to ensure that operations being performed actually get done, "eventually" (at some point in the future).

> SAGAs help ensure consistency and correctness across a distributed architecture (like a microservice-based one): a SAGA represents a high-level business process consisting of several low-level requests, where each one updates data within a single service. Each request has a Compensating Request that is executed when the request fails. However, not all actions can be completely undone, i.e. : an email that was sent to the wrong receiver cannot be un-sent. Nevertheless, we can *semantically* undo the action by sending another mail "Sorry, please ignore the previous mail". Compensating Requests semantically undo a request by restoring the application's state to the original state. Compensating requests are assumed not to fail, as this would imply a cascade of compensating requests.

We can implement the SAGA pattern in:

- **Event-driven choreography**, when there is no central coordination, each service produces and listens to other services' events to decide if an action should be taken or not.
- **Orchestration**: when a coordinator service is responsible for centralizing the saga's decision.

Event-driven choreography is based on:

- *SEC* (Saga Execution Controller): it attempts transactions in parallel or according to their "risk" (risk-centric approach). If an operation fails, then it can be recovered in one of two different modes:
    1. **Backwards**: If at least one transaction fails, we undo all successful transactions as well.
    2. **Forwards**: We retry all transactions until all transactions are successful.

If fault-tolerance is to be achieved, then operations ought to be idempotent. Such operations are simply retried until they succeed.

Figure 62: Architecture of the SAGA pattern: all operations are put in a queue; the SEC will pop an operation from a queue and will then handle it appropriately

The SAGA pattern is one of the two ways used to ensure eventual consistency:

1. Client determines when data is consistent (Dropbox approach with conflicting files)
2. Use fault-tolerant message queues and the SAGA pattern.

5.What is Eventual Consistency?

Eventual consistency is a relaxed transactional consistency policy, which does not mandate any type of transactional boundaries among multiple schemas and therefore allows to achieve ***high availability*** in distributed-transactions systems at the cost of a ***temporary inconsistent state***.

With eventual consistency, distributed databases are allowed to be in an inconsistent state temporarily, yet acknowledging the fact that it will reach a consistent state in the near future. Amazon, Netflix and Spotify make use of such consistency type. Customers are compensated adequately because of services' unavailability in the eventual consistency model. (Remember: Amazon AWS Refunds based on the downtime experienced by the user).

Such consistency type is profitable if the cost of compensating transactions is < the cost of enforcing a stronger consistency policy.

There are two main ways to ensure eventual consistency:

1. Client determines when data is consistent and is responsible for resolving conflicts (EX: Dropbox approach with conflicting files)
2. Use fault-tolerant message queues and the SAGA pattern.

6.What is an (event) data pump?

Data Pump

In monolithic applications, *reporting systems* usually run on read-only replicas of the main database for performance reasons, *pulling* the data for reporting purposes via projection operations in a database over multiple tables (SELECT operations with a certain condition).

In a distributed environment involving several services, pulling data involves performing API calls to several services, which may not be designed for reporting purposes, and would not be completely scalable; instead, pulling such large quantities of data would be detrimental for the performance of the webservice and the underlying database – because of HTTP calls' overhead involved in a large number of calls to different services.

With a ***data pump***¸ instead, we push the data to the reporting system's database directly. Alternatively, we could have a standalone program that directly accesses the database of the service that is the source of data, and pumps it into the reporting system. We hence understand that the *data pump*'s job is to map the database internal to a service to the database of the reporting schema. One way to control the data pump's functioning is via a `cron` programme, which controls how regularly data is pushed to the reporting system's database from each service's database.

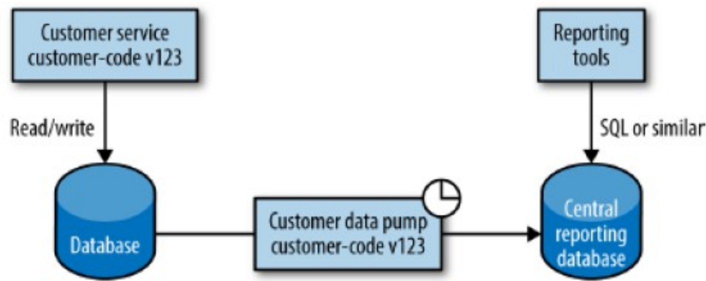


Figure 63: Scheme of a data pump's functioning

This architecture implies a tight coupling between the data pump and the service being accessed (as a rule of thumb, considering Version-Controlling the service and the pump together, and have the team developing the service be responsible for the data pump accessing that very service). NB: This coupling is still worth to pay, and some databases may offer ways of reducing this coupling.

Furthermore, we can piggyback backup operations over reporting, as both of these operations imply extracting large quantities of data from a database. [piggybacking means performing a backup operation at the same time of saving data for backup purposes → backup data would be saved in the reporting database too]

Event Data Pump (on multiple systems)

An event data pump is a data pump connected to a certain service, which emits (pumps) data when a certain event is emitted. For example, a customer service may emit an event when a given customer is created, or updated, or deleted: we may then write our own *event subscriber* that pumps data into the reporting database when a certain event is detected to be emitted.

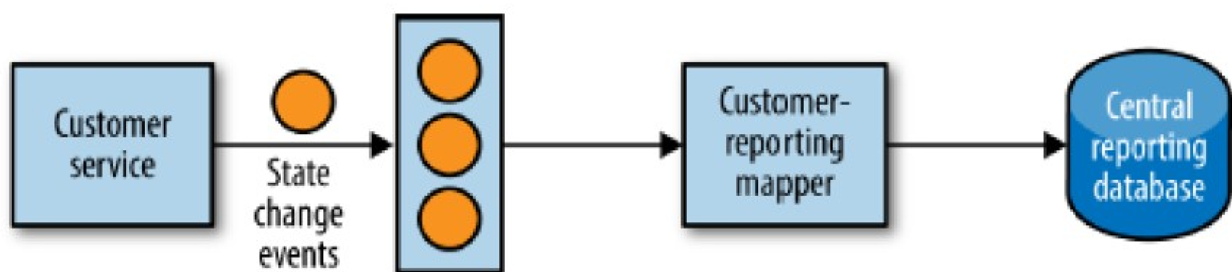


Figure 64: Architecture of an Event data pump, effectively decoupling a customer service from a reporting service by mapping events to the reporting system.

By inserting an *event subscriber (*which detects events being emitted), we effectively decouple the database of the source microservice from the one of the reporting database; also, as the coupling is reduced, the event data pump can now be managed by a separate team responsible for the microservice itself.

By storing already-processed events, we may just process the new events as they arrive, assuming the old events have already been mapped to the reporting system in an event-stream fashion.

The disadvantage lies in the fact that all the useful information needs to be emitted as events, and this approach may not scale well for large volumes of data.

Cloud-based Software Engineering

1. What are the service / deployment models of cloud computing?

The ***service models*** of cloud computing can be mainly divided into three categories:

1. **SaaS**: Software as a Service. The cloud provides software on-demand for use, accessible via light-weight clients or APIs. Applications are shared across multiple users, whose interaction is isolated from that of other users.
   The SaaS provider is responsible for all aspects of the deployment, from the infrastructure up to the (eventual) platform hosting the application and the application running, which is exposed to the SaaS client.
   The SaaS client is just a final consumer of the cloud-based services, who wants to benefit from the elastic scalability of the cloud without doing any software development, installation, configuration or maintenance.
   Analogy: go and eat your pizza at the restaurant
   EX: Salesforce, Dropbox, Google Drive, Facebook, Office 365
2. **PaaS**: Platform as a Service. The cloud provides a platform as a service, *enabling* the customer to develop an application via the cloud-based platform through a scalable and elastic runtime environment on demand, which hosts the execution of applications.
   The PaaS provider is responsible for managing the infrastructure, the OS and the enabling software. The enabling software (platform) is responsible for the scalability and the fault-tolerance, whereas the users are just required to focus on the logic of the application leveraging the provider's API and libraries.
   The PaaS client, a user who wants to leverage cloud computing for a scalable programming platform, creates his own application through the platform's interface, language or mechanisms made available by the PaaS provider, and is responsible for installing and managing the created applications.
   Analogy: Have the pizza delivered at your home.
   EX: Heroku, Azure platforms, Hadoop, Google AppEngine
3. **IaaS:** Infrastructure as a Service. The cloud provides on-demand virtualized servers, storage, and the networking infrastructure. It provides on-demand computing power in the form of virtual instances running on hardware.
   The IaaS provider is responsible for managing and maintaining the infrastructure.
   The IaaS client leverages the underlying infrastructure to build dynamically scalable computing systems requiring a specific software stack; he is responsible for the OS, the (eventual) platform running on top of the OS, the applications running on the OS.
   EX: Amazon EC2, S3.
   Analogy: Buy the pizza in the supermarket and bake it in the oven.

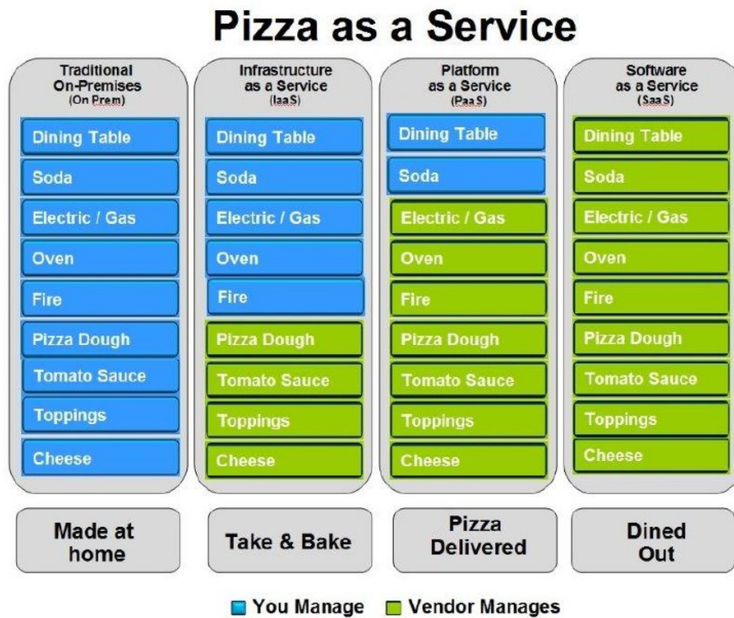


Figure 65: Analogy of the "Pizza as a Service" with the IaaS, PaaS, SaaS infrastructure

The ***deployment models*** of cloud computing are mainly three:

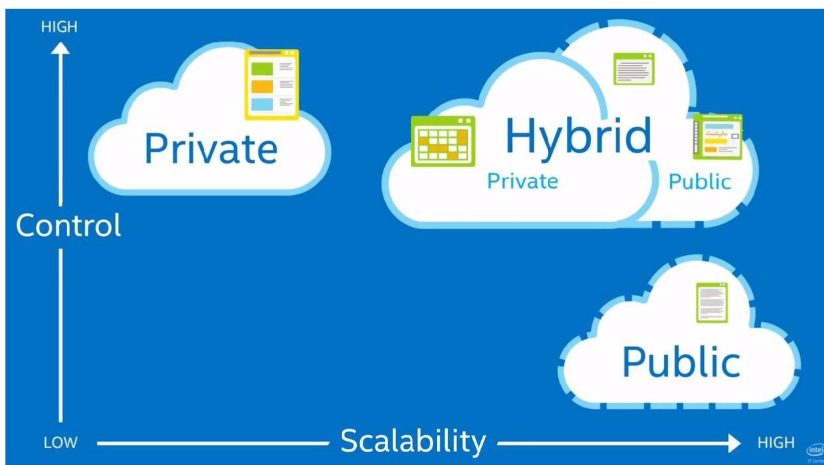


Figure 66: The three main deployment models of the Cloud: Private Cloud, Hybrid Cloud, Public Cloud

- ***Public cloud***: Most common deployment model for small enterprises or private customers, in which the necessary IT infrastructure (e.g: virtualized datacenters) is established by a 3rd -party service provider that makes it available to any consumer on a subscription basis. In this environment, users' data and applications are deployed on cloud datacenters on the vendor's premises. (EX: Dropbox / MEGA, Microsoft Azure)
    - *Low contro*l: control is delegated to the cloud provider. The customer / company can only trust the provider to maintain his data's safety and integrity.
    - *High scalability*: The provider manages big datacenters, exploiting the concept of "Economy of scale".
- ***Private cloud:*** A company owning hardware runs an on-premise cloud service on their own machines, replicating the cloud IT service delivery model in-house. The cloud service is only accessible by employees of the company. (Ex: Onedrive installed on a local machine). Such a

private cloud is employed when handling more sensitive data that needs to be managed in-house (e.g. in the case of governments or banks handling citizens' sensitive data).
- *High privacy and control*, because of the limited resource sharing and the non-visibility and accessibility of the service to the outer world.
- *Low scalability*: Because of the high CAPEX needed to buy hardware and the OPEX needed to maintain it, repair it, upgrade it.

- ***Hybrid***: A company has part of its data on a private cloud infrastructure and a part of its data on the cloud to meet the required users' QoS requirements. It is an in-between solution, combining the private and the public cloud's benefits. Such solution is usually adopted when wanting to handle and store more sensitive data locally, and less confidential data on the public cloud.
    - *Medium control:* The company just manages part of the infrastructure, with reduced CAPEX and OPEX wrt. a fully private cloud, and yet more control over its data.
    - *Medium scalability*: A medium CAPEX and medium OPEX needs to be disbursed respectively at the beginning of operations and to keep both the hardware as well as the cloud operational.

2.**Which is an example of (disruptive) business model exploiting cloud computing?**

Spotify – Freemium business model

An example of (disruptive) business model leveraging the use of cloud computing is *Spotify*. Spotify adopted a "freemium" business model, consisting of offering users the possibility to listen to music for free from any platform; whenever a users listens to a track, Spotify needs to pay a small fee to the royalty holders; revenue is provided by premium users' accounts, who pay 9.99€ for 3 months ad-free service and ads displayed to free users every 20-30 minutes.

The key idea underlying Spotify's success is the following:

"What if we provide free music to everybody?"

Originally, Spotify based its architecture off Amazon's AWS, but decided to move away from it, in what is known as "The Great Spotify AWS Cloud Exodus", in that Spotify decided to build its own infrastructure.

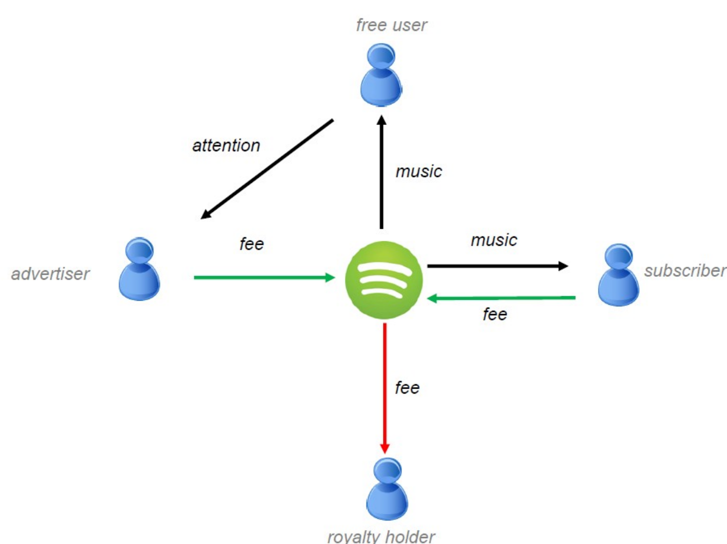


Figure 67: Overview of Spotify's business model. It pays royalty holders a very small fee for every single track played by its userbase. It gets revenue from advertisers, and from premium subscribers.

Dropbox – Freemium business model

An analogous freemium business model is the one adopted by *Dropbox*. They based their service on the idea:

"What if we provide free data storage to everybody?".

Free Users are able to store files in a limited quantity, but unlike in Spotify, they do not have to cope with annoying Ads. Premium users can enjoy a greater file storage capacity for business needs (1 TB of data storage or more).

Google – Customised Advertising business model

The Google Search Engine has made its fortune based on customized advertising: it tracks the interests of users based on their searches in order to profile them. When performing a search, or visiting a website, recommendations are provided based on an extremely large amounts of information collected for a user. Businesses pay for targeted ads, which cater to a niche of customers with interests matching the business' ads.

Google Search Engine's business started off based on the idea:

"What if we could provide a free search engine to everybody?

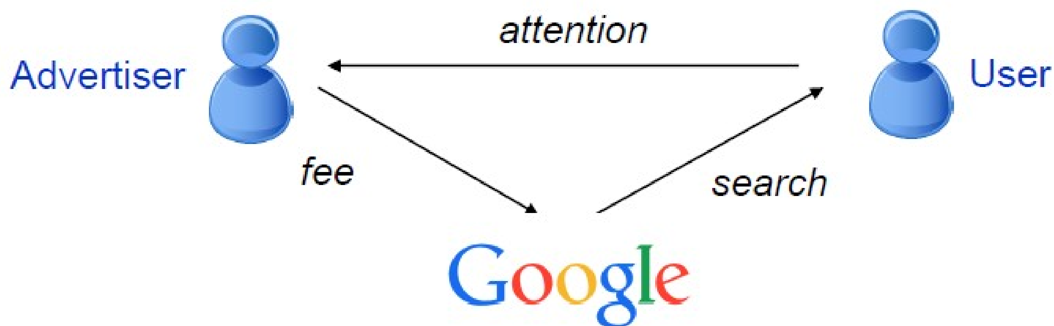


Figure 68: Overview of Google's business model

4. **What is a virtual machine? What is server virtualization?**

Virtual Machine:

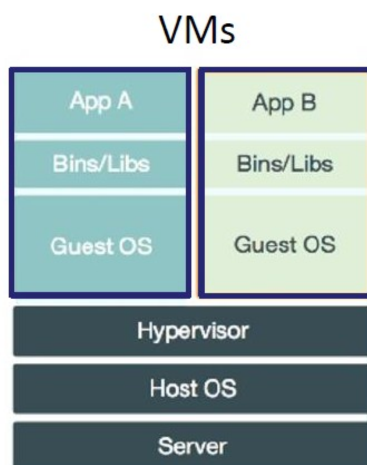


Figure 69: Schema of multiple Virtual Machines running on a Hypervisor

A virtual machine is an emulation of a computer system and its hardware. Virtual machines provide functionality of a physical computer via a combination of hardware and software. The most wide-spread type

of virtual machines are “*System virtual machines*”, which provide a full substitute for a real machine and its hardware, and the full functionality needed to run entire *Operating Systems*.

There also exist *process virtual machines,* which consist of an emulation of a normal application running in a host OS supporting a single process’ virtualization. This abstracts the underlying platform or OS, allowing a program to execute on a different machine in a seamless way (EX: Wine is an example of a process virtual machine, which allows to run Windows-based programs in a Unix environment).

Server virtualization

Server virtualization is a *Virtualization Layer* lying between the physical server (hardware) and the Operating System. It allows for the creation of different computing environments on top of the hardware.

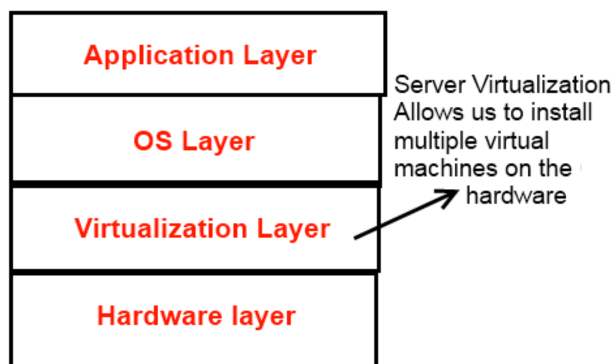


Figure 70: Layering visualization of where server virtualization lies at

A fundamental element of hardware virtualization is the ***hypervisor.*** The ***hypervisor*** is responsible for creating the virtualization layer itself. It contains the Virtual Machine Manager (VMM), a platform for managing Virtual Machines over the Hardware Layer. Hypervisors can be of two types:

- **Type 1 Hypervisor**: Low-level, loaded directly on top of the hardware, allowing multiple Operating Systems to run on the hardware. It it used in datacenters to make a certain OS run on hardware components (EX: AWS servers, decide the OS you want to use for your new server).
- **Type 2 Hypervisor:** Higher-level. They are loaded on top of an Operating System running on the hardware. They run on normal desktop / laptop devices and are generally used for creating an abstraction layer where an OS can be run on top of another OS, at the expense of a greater overhead wrt. Type 1 hypervisors (EX Type 2 Hypervisor: VMWare software for creating a Linux virtual machine on Windows or viceversa)

NB: Type 1 hypervisors are part of IaaS cloud services and are one key factor for the success of AWS.

4.What are Heroku’s Dynos for?

Heroku:

Heroku is a PaaS cloud platform based on a managed container system, with integrated data services and a powerful ecosystem for creating, deploying and running applications by combining functionalities provided by 3rd-parties addons.

It allows developers to build, run and scale applications seamlessly across different languages (Ruby, Java, PHP, Python, etc..).

Heroku Dynos’ Purpose

The *strength* of Heroku lies in Heroku Dynos. Heroku Dynos are containers which allow to run and scale Heroku apps. An application is made up of several different Dynos, which interact with one another.

Deploying an application onto Heroku (and relying on Heroku's dyno management) relives the developer from the burden of managing the underlying infrastructure and its configuration. Dynos' motto is:

"*Dynos make it easy to build and run flexible, scalable apps*"

Building / deploying Heroku Dynos

When deploying an app to Heroku, you package the app's code and its dependencies into ***containers*** (slugs in Heroku's slang). Similarly to Docker containers, ***Dynos*** are virtualized linux containers designed to execute user-specific commands in an isolated manner. A container is a lightweight, isolated environment providing computational power, memory, an OS and a barebone filesystem running on a shared OS. To deploy an app, Heroku needs only three things from the developer:

- Source code
- A list of dependencies
- A "Procfile" i.e: a text file indicating the command to start the application contained in the container

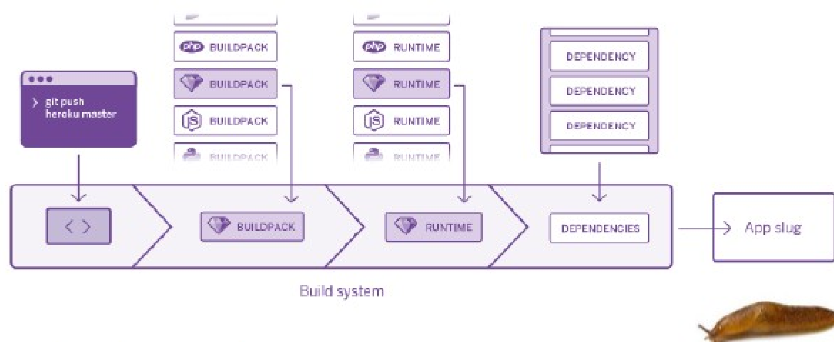


Figure 71: Overview of the Heroku build system

Build System: Receives the code from the container, fetches a build pack, language runtime and the code dependencies specified. Then it produces a ***slug*** app, which bundles the source code, the dependencies, the runtime in one single file ready to be run by the "***stack***", an Ubuntu OS maintained by Heroku.

The Heroku Dyno manager finally executes the resulting slug.

Dyno Type and their Functioning:

There can be several different types of Dynos' configurations in Heroku, namely:

- **Web Dyno**: The application redirects requests to a random Web dyno, which places it in a queue.
- **Worker Dyno:** Picks up the request and actually does the work, storing some data in a database eventually.
- **One-Off Dyno:** For performing one-off operations (i.e. : a database migration)

Heroku provides 150+ addons from third-parties for managing data services (EX: Redis, Postgres).

Scaling Heroku Dynos

Apps are able to *scale* in a *vertical* manner based on the amount of ***Dynos*** running for that application. Scalability can be managed in different ways and can even be achieved automatically if using proper (expensive) Dynos. When deploying their application, developers can choose one of different types of Dynos, with their own scalability type:

- Free, hobby
- Standard (h-scalability)
- Performance (h-scalability, autoscaling), for enterprise users

5.What is FaaS?

FaaS (Function as a Service) is a way to run code (i.e: a function) without having to fiddle with lower-level layers, like managing the OS or the hardware. For this reason, FaaS is also called “**serverless computing”:** you just provide the code and the platform will run it for you, hiding the low-level infrastructure and platform onto which the code runs. You are just billed by the computing time in a “Pay-per-usage” fashion

Some examples of FaaS are Amazon Lambda, IBM cloud functions or Google cloud functions. With AWS Lambda, functions can be designed via the AWS IDE or can be selected from a list of pre-built samples. In AWS Lambdas, potentially you can even setup your code to be automatically triggered by other AWS services or be called from other web apps when an event is generated. Other functions may subscribe to events and may even trigger some functions to store data in S3 buckets or in an external database. With Amazon Functions, we can monitor our app closely with real-time metrics.

6.How to avoid / reduce cloud lock-in?

What is vendor lock-in?

Vendor lock-in makes a customer dependent on a vendor for products or services after starting to use them, and unable to switch to another provider without incurring in substantial switching costs.

Cloud lock-in

Cloud service providers have a built-in lock in, not enforced by any contract as you just pay per usage, because of the following reasons:

- **Data Gravity**: It is hard to move large quantities of data from one cloud provider to another one.
- **Dependencies and compatibility issues**: It may be difficult to re-deploy an application to a different infrastructure / platform. Especially when using Add-ons, you become dependent on one cloud provider.

According to *Thorsten’s Lock In hypothesis*: the higher the cloud layer you operate in, the greater the lock-in.

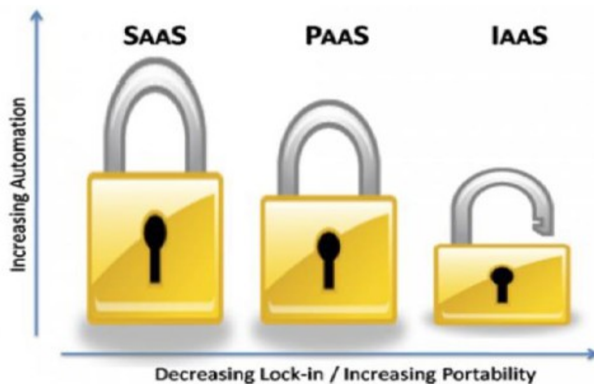


Figure 72: Thorsten's Lock in Hypothesis

How to avoid / reduce lock-in:

Ideally, we would like our applications’ cloud components to be hot-swapped with other providers’ components. Unfortunately, the hard reality is well different from this. To reduce or avoid the impact of lock-in, we may adopt one or multiple approaches:

Commercial principles:

- Use different cloud providers, but don’t commit to a single one of them
- Consider carefully about using proprietary cloud-vendor services, which you may be unable to change
- Plan an “exit strategy” from a cloud provider

Technical principles:

- Ensure portability of data
- Use unified interfaces
- Go for open standards and open source-technologies (Amazon: WE are the standard, 50% of the Cloud market)

DevOps principles:

- Use loosely coupled architectures, where a certain part of the application can be easily replaced. Integrate API/Rest
- Use microservice-based architectures, where different microservices can even be deployed to different cloud providers
- Use containers and devops tools

7.What is a container / image / volume?

Container

A container is a lightweight, isolated environment providing computational power, memory, a light OS and a barebone filesystem running on a shared OS. A container is a running instance of an image. Containers are created from images, inherit filesystems from images and use images' metadata to determine the startup configuration.

With an analogy to the Object-Oriented world, images can be thought of as objects (I.e: a dynamic instantation of classes)

Image

An image of a container is a static collection of *filesystem layers* and some *metadata*. The metadata has information about environment variables, port mappings, volumes, etc…

Image files take up a lot of space on disk because they must have their own copy of every single dependency specified (i.e: language-specific libraries, tools, environments).

An image can be considered a read-only static snapshot of a container. A running instance of an image is a *container*. By default, images are read-only and do not support data persistence across container instances.

Images can exist for virtual machines as well: in that case, images would emulate both the hardware and the OS. (EX: Image of Ubuntu 16.04 ready to be started by VMWare).

With an analogy to the Object-Oriented world, images can be thought of as static classes.

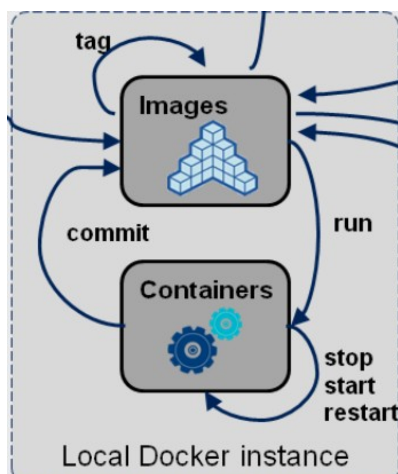


Figure 73: Overview of images and containers in Docker

Volume

A volume is a single accessible storage area within a file system, accessible via an operating system's logical interface. A file system is an intermediate layer controlling how files are stored in an operating system.

External volumes are the preferred mechanism for persisting data generated by and used by Docker containers. Once a volume is mounted, it allows for data persistency in a container.

All files created and modified within a container inside a container are stored on a writable container layer in a *copy-on-write* fashion, maintaining isolation from other containers' files at the same time. Copy-on-write is a standard optimization strategy used in computing: when creating a new object from a template, you only copy data over when it's changed, yet not affecting the original file (which may be shared by multiple instances).
→ Whenever a running container needs to write to a file, it records the change by copying the item to a new area of disk. When a docker commit is performed, the new area on disk is frozen and is recorded in a layer with its own identifier in the image.
[When multiple running containers commit to one image, some Git-like mechanisms should be put in place to resolve eventual conflicts]

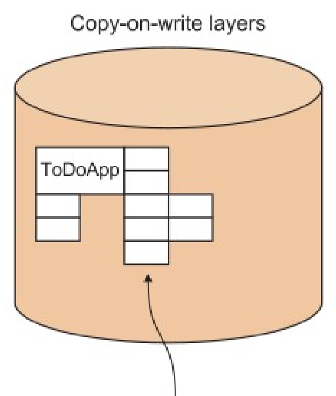


Illustration 4: Overview of the copy-on-write mechanism for handling files' modification

8.What is the difference between a virtual machine and a container?

A ***container*** runs natively on Linux and shares the kernel of the host machine with other containers. It takes no more memory than any other executable, making it lightweight. For this reason, containers are very fast to start via a *container engine* and simpler to build wrt. a VM, as just a lightweight OS with a filesystem (and eventually a volume) needs to be started; the first time a docker image is ran, it may need to resolve its dependencies by fetching them from the Docker Hub. However, containers are less secure and provide least isolation wrt. a VM.

A ***virtual machine*** runs a full-blown "guest" OS with virtual access to host resources. In general, VMs provide an environment with more resources than most applications need and for this reason they are way

slower to be started than containers, yet provide a greater deal of security. Also, Virtual Machines can operate at the hardware level already (See Type-1 Hypervisors), whereas a container engine must operate starting from a shared OS.

Containers are so fast to be ran because they contain all the information in the Image file, which was specified in the dockerfile or committed previously.

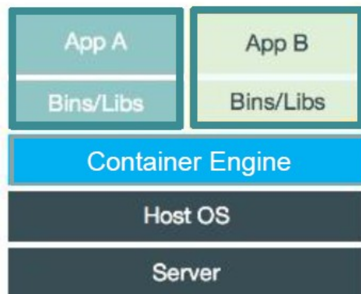


Figure 74: Schema of multiple Virtual Machines running on top of a Type-1 hypervisor

Figure 75: Schema of a container engine

running on top of a host OS, and hosting multiple containers

9.What is image layering in Docker?

Image layering in Docker is a fundamental concept: every container requires a set of dependencies (let them be an OS, a filesystem, some static libraries). What if you have 100s containers running on a machine? By default, each one would need to have their own dependencies replicated on disk to run in an isolated environment → Our disk space would run out pretty quickly!

The solution adopted in Docker is based on *static layering*; a docker container image is made up of several different layers, which are specified in a dockerfile and are stacked one of top of the other. Each layer except the very last one is read-only.

You can just build your existing application of top of existing layers and apply changes at higher layers. All layers can be shared across multiple running containers, much like a a shared library can be shared in memory across multiple running processes (this feature is essential to avoid replicating images whenever starting a container, as this could lead to running out of disk space).

One further optimization is represented by the copy-on-write mechanism specified in question (8).

One ***layer*** consists of one part of the application: it depends on underlying layers (dependency layers), whereas it provides *abstraction* for layer above. (much like a computer or network architecture).

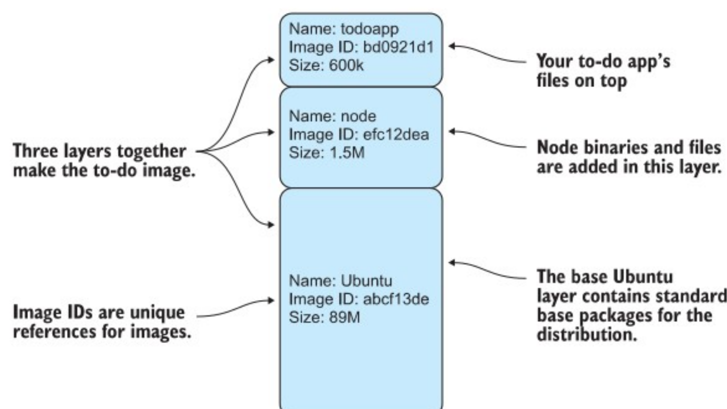


Illustration 5: Overview of the filesystem layering of an image.

Layer1: Ubuntu image

10.What is the effect of docker run/commit?

```
docker run <image_name> <command_to_execute>
```

The `docker run` command has the effect of launching a container with a specified image name (a lightweight distribution of a Debian Linux Distribution) and a certain linux command. If the image specified is not present locally, then it is searched on the DockerHub and eventually downloaded. Upon downloading the image, the image is instantiated and the command passed is run on the newly created instance. Once the command specified is terminated, the container is shut down. Optionally, the docker instance can be set to remain up even after the command execution terminates.

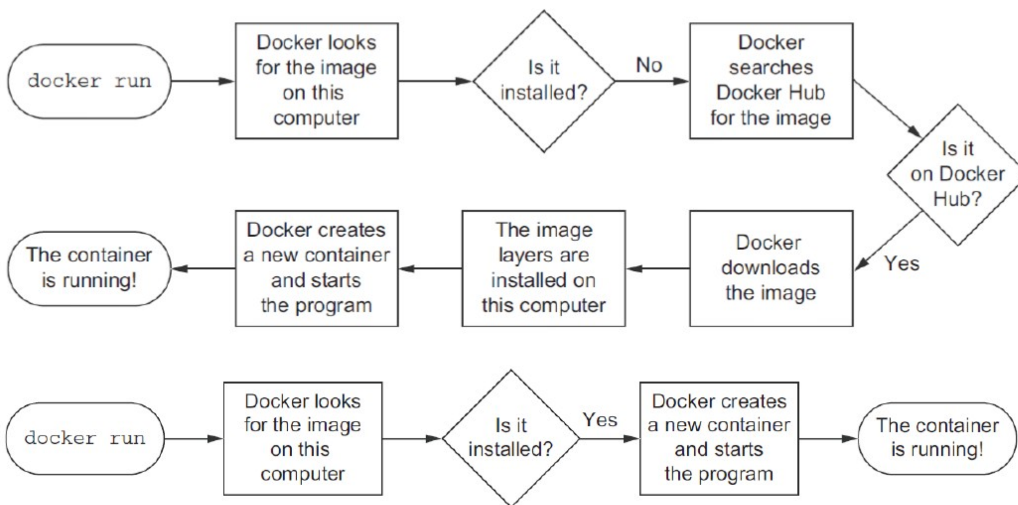


Figure 76: Schematic representation of the process to run a docker image

```
docker commit <running_container_name> [repository_name]
```

A docker commit is just the opposite of a docker run command. It creates a Docker image based on a running Docker container by "exporting" the changes applied within a container to a static image (EX: Say, you modified some files in a local container locally and may want to push these changes to the image before deploying in production, you may do so via a docker commit). Changes are saved to an image based on the "copy-on-write" mechanism.

Business Process Modelling

1.What is a parallel / exclusive / inclusive gateway in BPMN?

BPMN

BPMN stands for Business Process Modelling and Notation. It is a notation used to specify business processes in an unambiguous, formal and abstract way, mutually intelligible by programmers, businessmen and customers. It is used for easing the communication between different stakeholders working on a process according to the principle "A picture (or a scheme) is worth a thousand words".

Gateway:

A gateway in BPMN is a decision point that can adjust the path of a flow based on certain conditions.

> In Gateways, there are some gating mechanisms that either allow or disallow passage through the Gateways. As tokens arrive at a Gateway, they can be merged together on input and/or split apart on output.

Parallel Gateway in BPMN (AND)

A parallel gateway in BPMN is a decision point,
triggering two output flows to continue their execution on both branches in *parallel*. The two branches are then joined at a ***join point***, when the parallel execution of the two branches finishes.

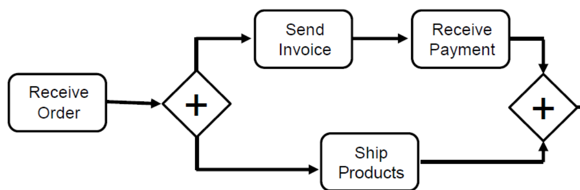
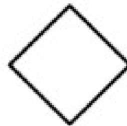


Figure 77: Parallel gateway and parallel join point

Exclusive Gateway in BPMN (XOR)



An exclusive gateway in BPMN is a decision
point, triggering *one single output execution flow* out of the different output branches, based on the branches' conditions being met. I

The two branches are then joined at an exclusive join point.

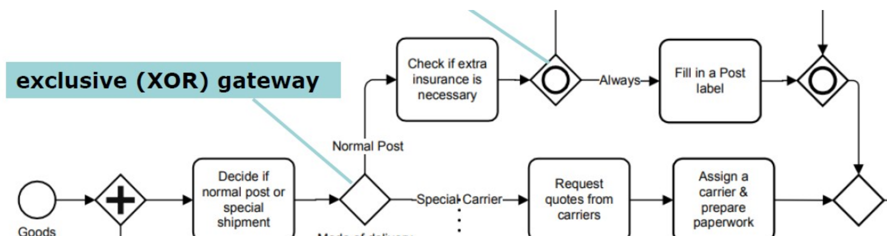


Figure 78: Exclusive Split point and Exclusive Join point in BPMN

Inclusive Gateway in BPMN

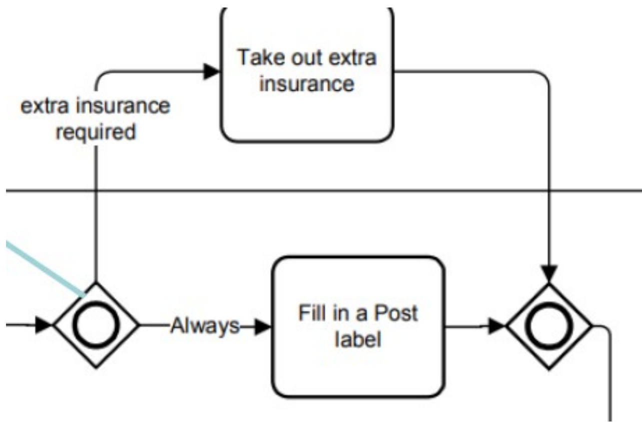
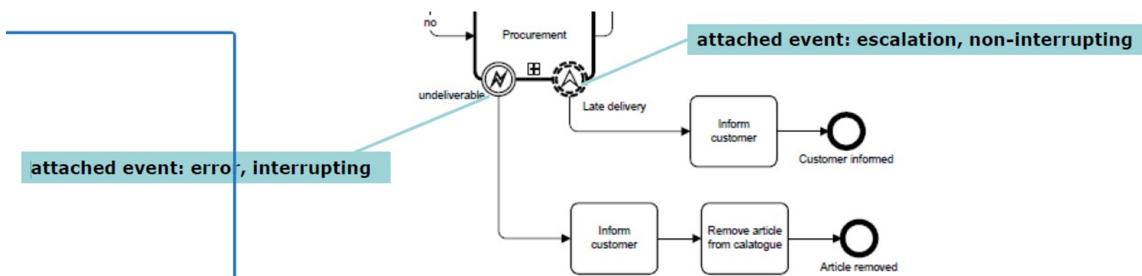


Figure 79: Example of an Inclusive Gateway in BPMN. The first inclusive gateway is a split, the second one is a join

An inclusive gateway in BPMN is a decision point triggering *one or multiple execution flows* out of the different output branches, based on the branches' conditions being met.

2.What is an error event in BPMN?



An error event attached to the boundary of an activity (either a task or a subprocess) is going to catch an error if that very activity is being processed while the error is thrown. If the error is thrown after the activity has been completed, then the error will not be caught.

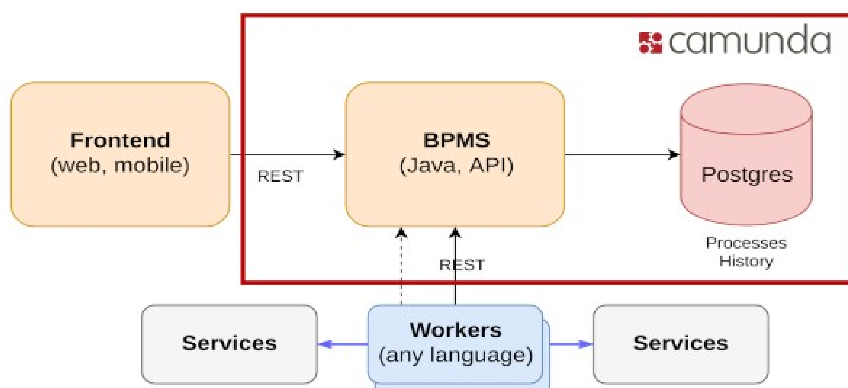
Events can be distinguished into two different types, based on the border line they have:

- **Interrupting Event**: solid black border line with white padding color in-between. (EX: error event in the Figure)
- **Non-interrupting Event**: dashed border line with white padding color in-between. (EX: escalation event in the Figure).

3.What is Camunda?

Camunda is a light-weight, open-source platform for Business Process Management. It is a framework supporting BPMN for workflow and process automation. It provides a RESTful API, which allows you to use your language of choice when interacting with Camunda.

Camunda can be used to describe, deploy and test Business Process Models via a Graphical User Interface through the Camunda Modeler. Deployed models via the Camunda Modeler can be run via an Apache-based web application.



4. Can you describe the usage patterns of Camunda?

----See the LAB slides of ASE---

5. What is the difference between orchestration and choreography?

Choreography consists of different processes interacting with one another with no central coordination (EX: Different dancers with no need for a dance director: each one knows what they have to do and what their role is in the show). Initially, Netflix' microservices cooperated with one another via choreography in a de-centralized way via P2P communication; following some negative experiences with this approach, Netflix moved away from this approach and went for a centralized orchestration approach.

Orchestration consists of coordinating different processes or tasks via a centralized architecture (EX: in an orchestra, the orchestra director directs the musicans playing their music). Netflix adopts the same centralized approach to microservices' coordination.

6. What is a workflow net?

When modelling Business Processes, we would also like to prove properties of the models created. For this purpose, we may want to make use of Petri Nets and Workflow nets.

A ***workflow net*** is an extension of Petri nets. Petri nets consist of places (process states), transitions (process transitions to other process states) and directed arcs connecting places and transitions. No transitions can subsist between places and places or transitions and transitions.

Transitions model BPMs' activities; *places* and *arcs* model execution constraints. System dynamics are represented by *tokens*. The presence of tokens in certain places determines the state of the system. A transition can fire if there is a token in each one of its input places.

A petri net is a workflow net iff the following ***three requirements are satisfied***:

1. There is an initial *place* with no incoming edge (nothing goes in at the initial place)
2. There is a final *place* with no outgoing edge (nothing goes out at the final place)
3. All places and transitions are located on some path from the initial place to the final place (i.e: all places and transitions can be visited over some path).

Workflow nets are used as an abstraction of the workflow, which is used to check the so-called soundness property. This soudness property guarantees the absence of live-locks and deadlocks.

7. How can we model BPMN parallel/exclusive/inclusive gateways with workflow nets?

Parallel gateways in workflow nets:

A parallel gateway can be modelled through an AND split and an AND join in workflow nets.

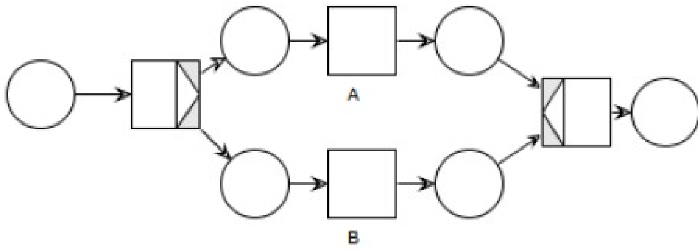


Figure 80: Example of a parallel gateway modeled in a workflow net via an AND gateway

Exclusive gateways in workflow nets:

An exclusive gateway can be modelled in workflow nets through a XOR split and a XOR join.

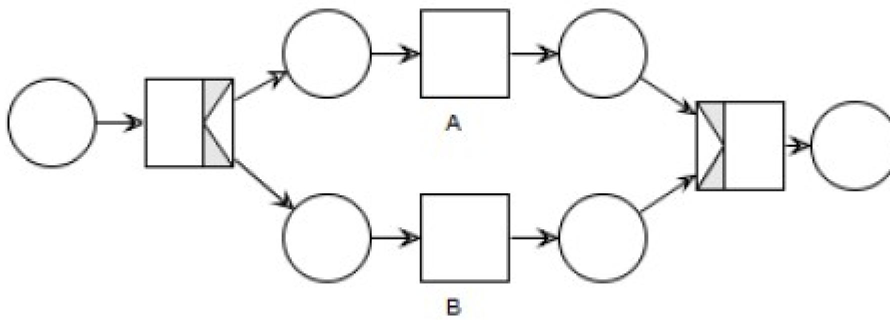


Figure 81: Example of an Exclusive Gateway modeled via a XOR split and a XOR join in a workflow net

Inclusive Gateway

An inclusive gateway, allowing multiple transitions to occur based on their conditions being verified, is harder to model in a workflow net:

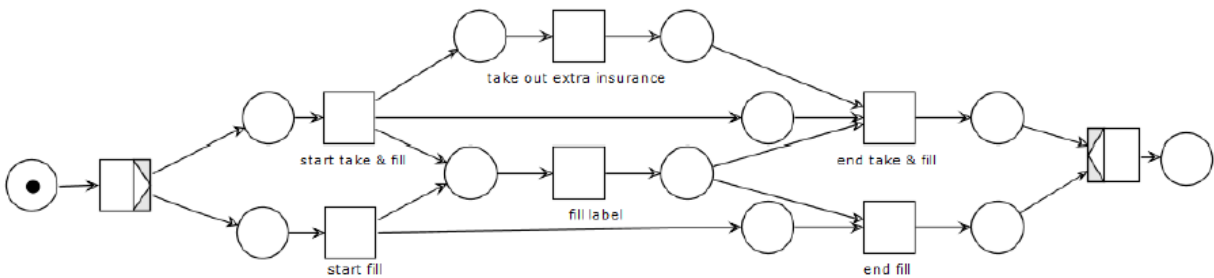


Figure 82: Example of an inclusive gateway in a workflow net

8. What is a live / bounded / sound net?

Live petri net

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 *live* iff: $\forall M' \wedge t, \exists M'' reachable M', which enables t$

(i.e: at every state's transition, there is a state which allows the transition to be triggered).

Bounded petri net

A petri net is *bounded* iff for each place *p* , there is a natural number *n* such that for every reachable state *s*, the amount of tokens in p is less than *n*.

A petri net P is bounded iff: $\forall p \exists n \ni N : \forall s, T(p) < n$

(i.e: the amount of possible tokens in a certain place is bounded by a natural number *n).*

Sound workflow net

A workflow net N is *sound* iff (Ň, i) is live and bounded, where Ň is N extended with a transition from the final place *o (output)* to the initial place *i (input).*

NB: We may have a live, but not bounded petri net.

We may have a bounded, but not live petri net.

We have may a live and bounded petri net (but not a sound workflow net), if there is no transition between the final place o and the initial place I.

Fog Computing

1.What is Fog Computing?

Fog Computing is a concept directly spurring from the rising of the Internet of Things (IoT); The Internet of Things could enable innovation that enhances the quality of life in a number of sectors, but it generates a sheer amount of data that is difficult for traditional systems and infrastructure to process (i.e: the Cloud, with its scalable architecture), mainly due to network bandwidth bottlenecks and latency requirements of latency-critical system.

Fog computing is designed to overcome such limitations by filtering and processing the data before the cloud, obtaining a reduced latency via a partial *in-situ* computation, which reduces the amount of data needing to be sent to the cloud and hence the strain on the network bandwidth.

Figure 83: Edge (Fog) Computing: computation is moved towards the IoT device, obtaining reduced latency, but limited computational power; Cloud computing: high latency, but potentially infinite computational power

Edge Computing

To address the bandwidth bottlenecks and the latency issues of sending data to the cloud, edge computing was introduced. Edge Computing consists of bringing the processing closer to the source (i.e: IoT sensors) for local storage and preliminary data processing, to achieve a reduced latency and get rid of the bandwidth bottleneck in the networking infrastructure, accelerating the final decision-making process (thanks to the

reduced latency). However, Edge devices cannot handle multiple IoT applications competing for their limited resources, which result in resource contention and increased processing latency at the edge devices.

Fog Computing – Definition:

*«A system level horizontal architecture that distributes resources and services of **computing**, **storage**, **control** and **networking** anywhere **along the continuum from Cloud to Things**, thereby **accelerating** the velocity of **decision making**.»*

Fog computing seamlessly integrates edge devices and cloud resources, helping to overcome limitations of pure cloud-based systems and pure-edge devices. It avoids resource contention by leveraging cloud resources and coordinating the use of geographically distributed edge devices in a facilitated and standardized way.

It is a distributed paradigm that provides cloud-like services to the network edge by utilizing clients or edge devices near users to carry out a substantial amount of the computation, communication, storage, control, configuration and management. The approach benefits from edge devices' close proximity to sensors, while leveraging the on-demand scalability of cloud resources.

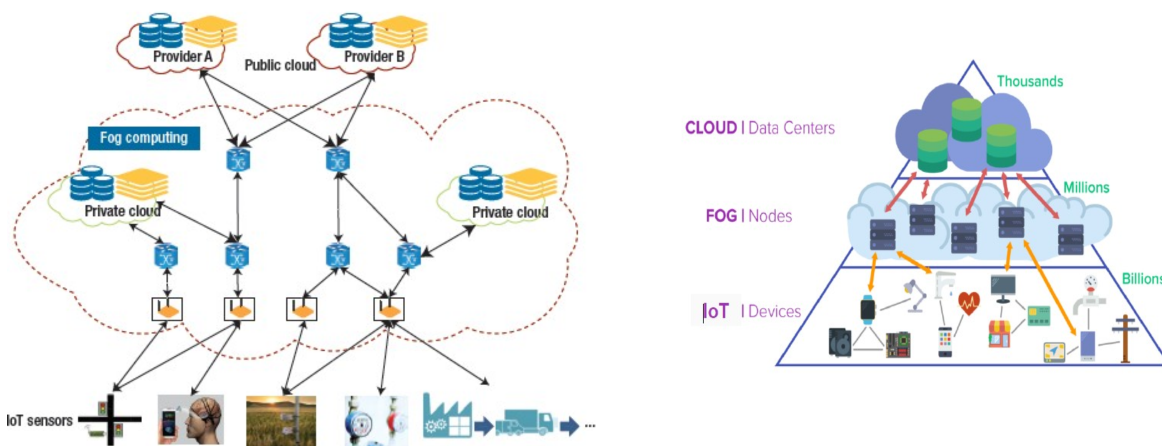


Figure 84: Fog Computing Architecture supporting edge devices by leveraging cloud computing

Fog Computing Characteristics

- Heterogeneity of devices (sensors, actuators, cars, TVs, ovens, laptops, …)
- Collaboration & interoperability between different devices
- Pervasiveness & geo-distribution (IoT alike)
- Low-latency & bandwidth saving (by placing the processing closer to the IoT device)
- Fog & Things mobility (autonomous cars, trains, airplanes)

2.What is / how difficult is the "component deployment problem" in fog computing?

The component deployment problem in fog computing consists of deploying different components of a fog-computing application over a fog infrastructure, respecting its requirements wrt. QoS (ex: bandwidth / uptime) and hardware cost. When deploying a fog application over a fog infrastructure, we may need to find answers to a series of questions, namely:

- *How many and how powerful fog nodes do I need to* ***adequately deploy*** *my application?*
- *Should I deploy this component on the* ***cloud****, on the* ***fog-as-a-service*** *available in my city or on my* ***premises gateway****?*
- *Is there any component I should better deploy on a different node after this* ***link/node failure****?*
- *Which are the eligible deployments that* ***comply most*** *with the required* ***qos****?*
- *Can I* ***reduce*** *resource consumption of some fog nodes, or* ***avoid*** *using them?*

Figure 85: Questions that need to be answered when deploying a fog-computing application over an infrastructure

Namely, this involves finding a mapping from a set of components we want to deploy to a set of physical fog nodes (which make up the infrastructure), and trying all possible *deployments* of components onto the infrastructure, at the same time respecting the QoS and cost requirements. Such problem is known as the "**CDP**", Component Deployment Problem.

The Component Deployment program involves trying all possible deployments of a fog application's components and finding one that optimizes the metrics and QoS specified. It is an NP-complete problem, as the CDP problem can be transposed into the *graph-isomorphism problem* (is a graph a subgraph of another graph □ how do we map all components to all fog nodes at our disposal?).

By showing that the CDP problem can be transposed into a graph isomorphism problem, we are able to show that our problem can be re-mapped into an NP-Hard problem.

Time: O(e^c ¿ □ Exponential complexity!

Space: Polynomial, if remembering previous combinations.

3.What is FogTorchΠ ?

Modern applications usually consist of many independently deployable components (each with its own hardware, software and IoT requirements) that interact together in a distributed way. Such interactions may have stringent QoS requirements – latency, bandwidth – to be fulfilled for the deployed application to work as expected. Thus, when deciding where to deploy application components, one should check their hardware, software, IoT and QoS requirements against the offerings of the available infrastructure.

> Tools to support app deployment to the Fog should desirably feature (i) QoS-awareness to achieve latency reduction, bandwidth savings and to enforce business policies, (ii) context-awareness to exploit local and remote resources, and (iii) cost-awareness to enact cost-effective deployments.
>
> FogTorchΠ It is an open-source *java* prototype that permits to describe IoT+Fog+Cloud infrastructures and applications to determine QoS, context-aware and cost-aware application deployments. So, it's an implementation of the CDP, which it solves via backtracking. FogTorchPI estimates the QoS of the links through ***Monte-Carlo simulations*** 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
> varies according to a certain probability distribution.

The CDP takes as input the following parameters:

Input:

- A multi-component application ***A*** with requirements ***R***
- A distributed Fog infrastructure ***I*** (nodes and links)

- A set of objective metrics **M**

Output:

The best application deployments (Delta) that meet all requirements in **R** and optimize metrics in **M** (i.e: QoS / Cost).

FogTorchPi solves the CDP via backtracking in exponential time by brute-force trying all combinations. It estimates the QoS assurance by varying links via **Montecarlo** simulations and estimates the **Fog resource consumption** and the **monthly deployment cost** ($$$).

4.What is SecFog?

SecFog is the implementation of a methodology to assess the security level of multi-component application deployments in Fog scenarios. It can be used in combination with FogTorchΠ: after finding an eligible deployment with FogTorchΠ, we would still like to evaluate this deployment's security level.

The methodology designed for assessing the security level of a deployment is structured as follows:

(We assume each node self-describes its security capabilities and their reliability through a Node Descriptor using the taxonomy vocabulary.

a. A ***taxonomy of security capabilities*** in Fog Computing was identified, as no Security Control Framework currently exists for the Fog (only for the Cloud, according to ISO/IEC 19086, EU Cloud SLA Standardisation Guidelines)

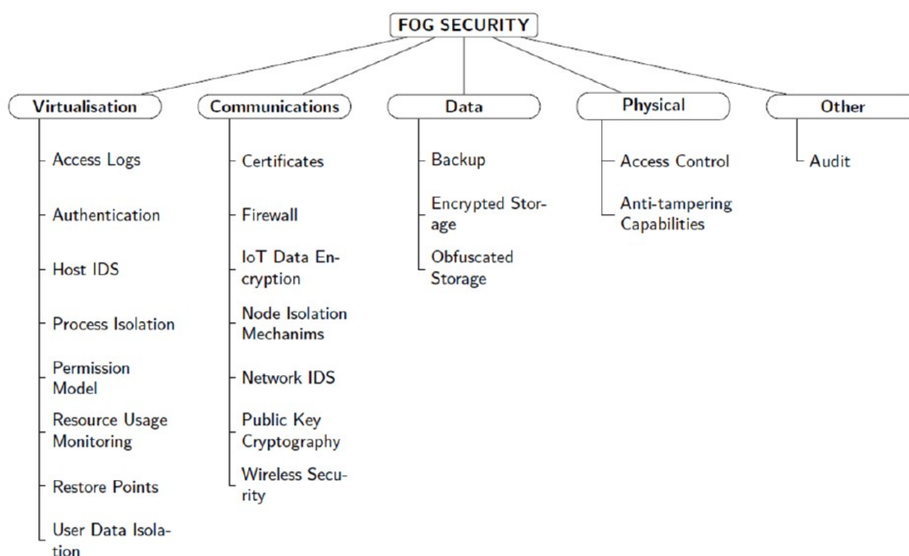


Figure 86: Taxonomy of Security Capabilities defined for Fog Computing in SecFog

b. Specify the **Security Capabilities:** Infrastructure operators are responsible for maintaining their infrastructure. Infrastructure operators describe the infrastructure node's security capabilities (and their reliability) through a Node Descriptor (ND) using the taxonomy vocabulary.

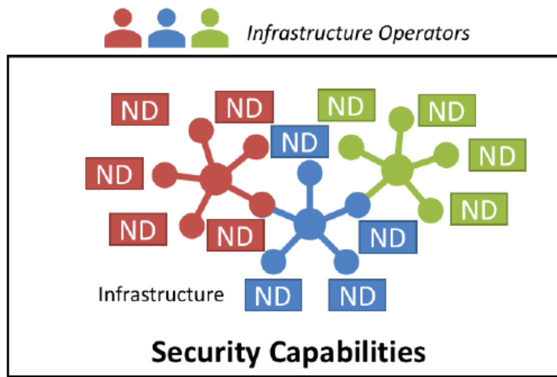


Figure 87: Security Capabilities of an infrastructure

c. Specify the **Security Requirements** and Specify the **App topology:** App operators are responsible for managing the application. They describe their app topology and annotate it with security requirements both at the ***component level (CR – Component Requirements)*** and ***application level (AR – Application Requirements)*** based on the taxonomy dictionary.
d. *The App operator* (can) also specify the complete or partial **App Deployments** of their application, as well as the **Trust degrees** towards different Infrastructure Operators.

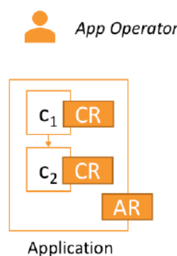


Figure 88: Application topology

e. The AR and CR can then be expressed in terms of Custom Security Policies, and Custom Security Policies are expressed in terms of the **Default Security Policies**

The prototype implementation was made in Problog2 (a probabilistic programming language), which matches application security requirements with infrastructure capabilities. (generate & test – Prolog-like). What it does is:

1. It matches application security requirements with infrastructure capabilities (generate & test) approach.

2. It obtains the security level of each deployment by multiplying the reliability of all exploited security capabilities weighted by their *trust degrees (omega_m)* .

The deployments obtained are ranked according to the following formula for ***multi-objective optimization***

$$r(\Delta) = \sum_{m \in M} \omega_m \times \overline{m(\Delta)}$$

Illustration 7: Multi-objective optimization. M = Set of metrics to be optimized; m(Δ) is the normalised value of the metric m for the deployment Δ. Metrics = where QoS assurance, resource usage, cost and security are optimized.

Deployment: A deployment can be recursively defined as the association of each component C to a node N.

Final Output:

The *output of FogTorchPi* consists of a deployment where the QoS assurance and the security are maximized, whereas the cost is minimized.

→ A list of *deployments* with an associated security level ranked according to such metrics is produced. We might either go for fog-ward or for cloud-ward solutions, where the fog and cloud resources are maximized, respectively.

The security level of one deployment is obtained by multiplying the reliability of all exploited security capabilities, weighted by the trust degree in the operator where that part of the application is deployed.

Example of a Component Security Requirement:

As an example, let's consider a data storage component we want to deploy:

The security requirements of the data storage component can be specified as:

```
secure_storage(N) :-
    backup(N),
    (encrypted_storage(N); obfuscated_storage(N)).

secureComponent(data_storage, N, D) :-
    secure_storage(N),
    access_logs(N),
    network_ids(N),
    secure(data_storage, N, D).
```

Example of an app Security requirements:

Finally, app-wise security requirements

Public Key Cryptography and Authentication

required app-wise

can be specified as:

```
extras([]).
extras([d(C,N)|Ds]) :-
    public_key_cryptography(N),
    authentication(N),
    extras(Ds).
```