# 1. Software Products

The old way was to develop a project commisioned by a customer. The starting point was a set of *software requiements* that are owned by the external client.
Requirements alwayas change so the contractor had to change the software to refelct those changes.
**Custom software usually has a long-lifetime.**

**Must Businesses don't need customised software**

Another paradigm is **Product-Based Software Engineering.**
The starting point for product development is a buisness opportunity that is identified by individuals or a company, which develop the software product to take advantage of this opportunity, and has responsability for deciding the development timescale, what features to include and when the product should change.
**Rapid delivery of software products is essential to capture the market for that type of prodcut.**

**Product-Based Software Engineering:**

- developers identify an opportunity
- the opportunity inspires a set of product features to develop
- the features are implemented in the software product
- the software product realizes and satisfy the envisioned opportunity
  The only and main actor in the process are the developers.

## Product Vision

Product Vision are simple statements that define the essence of the product to be developed.
A PV answers the three foundamental 3W questions:

1. what is the product?
2. who are the target customers?
3. why customer should buy the product?

**The product vision is determined by DPCP Experience:**

- **domain:** the developers work in a particular are and understand the software support they need
- **product:** users of the existing software may see simpler and better ways of providing functionality
- **customer:** developers may have extensive discussions with prospective customers

- **prototyping:** prototyping and playing around to understand better the product and technologies

**PRODUCT VISION GENERAL TEMPLATE:**

- *for* (target customers)
- *who* (statement of the need or opportunity)
- *the* (product name) is a (product category)
- *that* (key benefit)
- *unlike* (primary competitor)
- *our* product (statement of primary differentitation)

## Software Product Management

**Software product management is a business activity that focuses on the software products developed and sold by the business.**
Product Managers take overall responsibility for the product and are involved in planning, development and product marketing.
**Product Managers are the interface between the organization, its customers and the software development team.**

**Product Management Concerns:**

- business needs: the product meets the business goals
- tech constraints: developers have to be aware of tech issues that are important to customers
- customer experience: PM should be in regular Contact with customers and potential customers

**Technical Interactions of PMs are:**

- product roadmap
- user stories and scenarios (to identify product features)
- product backlog
- acceptance testing (verify that the release meets set goals)
- customer testing (to get feedback)
- UI design (to monitor simplicity of use)

## Product Prototyping

**Product Prototyping is the process of developing an early version of a product to test your ideas and to convince yourself and the company funders that your product has real market - potential.**

- critically important to demonstrate software to potential customers and funders

- can help revising design
- aim at having first -reduced- prototype quicky (within 6 weeks)

# 2. Agile Software Engineering

**Agile Software Engineering focuses on delivering functionality quickly, responding to changing product specifications and minimizing development overheads.**
**These methods focus on the software rather than its documentation.**
The software is developed in a series of increments and aim to reduce process bureaucracy as much as possible.

**The Agile Manifesto:**

- individuals and interactions > processes and tools
- working software > comprehensive documentation
- customer collaboration > contract negotiation
- responding to change > following a plan

**The Agile Method Process can be synthesized by following these 12 principles:**

1. the highest priority is satisfy the customer through early and continious delivery of valuable software
2. welcome changing requirements, even in late development
3. deliver working software frequently
4. business people and developers must work together
5. build projects around motivated individuals
6. the most efficient method of conveying information is face-to-face conversation
7. working software is the primary measure of progress
8. agile processes promote sustainable development
9. attention to technical exellence and good design
10. simplicity, the art of maximizing the amount of work not done, is essential
11. the best emerge from self-organizing teams
12. at regular intervals, the team reflects on how to become more effective

## Extreme Progamming

**Extreme Programming focused on 12 new development techniques that were geared to rapid, incremental software development, change and delivery:**

1. **incremental planning/user stories:** there is no grand plan. Instead what needs to be implemented in each increment are established in discusion with a custome representative
2. **small releases**: the minimal useful set of functionality that provides buisness value

3. **test driven development:** instead of writing code then tests for that code, developers write the test first. This helps clarify what the code should actually do and that there is always a tested version of the code available
4. **continuos integration:** as soon as the work on a task is complete, it is integrated in the whole system
5. **refactoring:** all developers are expected to refactor the code as soon as potential code improvements are found

# Scrum

**Scrum is a lightweight framework that helps teams to generate value to adaptive solutions for complext problems.**

Scrum is founded on **empiricism**, it asserts that knowledge comes from experience and making decisions based on what is observed, and **lean thinking:** it reduces wase and focuses on the essentials.

**Scrum Theory Pillars: TIA**

- **transparency:** the emergent process and work must be visibile to those performing the work as well as those receiving the work
- **inspection:** the progress toward the goals must be inspected frequently to detect potentially underirable variances
- **adaptation:** if any aspects deviates outside acceptable limits or if the resulting product is unacceptable, the process being applied or the mateirals being produced must be adjusted

## Scrum In Practice

**The scrum's main components are:**

1. **the team:** devs, product owner and scrum master
    1. **the product owner** is responsible of ensuring that the development team is always focused on the product they are building rather than diverted into technically interesting but less relevant work. In product development the product manager should normally take the product owner role
    2. **the scrum master** is a scrum expert whose job is to guide the team in the effective use of the scrum method
2. **artifacts:** product backlog, sprint backlog
3. **scrum events:** sprint planning, execution and review.

### Product Backlog

**The product backlog is a todo list of items to be implemented that is reviewed and update each sprint.**

The items on this list are called Product Backlog Items (PBIs) and PBIs are ordered by priority.

**PBIs types:**

- feature
- user request
- dev activity
- engineering improvement

**PBIs states:**

- **ready for consideration:** high-level ideas and feature descriptions
- **ready for refinement:** the team has agreed that this is an important item that should be implemented, there is a reasonably clear definition but more work is needed to understand and refine the item
- **ready for implementation**

**Product Backlog Activities:**

- **refinement:** existing PBIs are analyzed and refined to create more detailed PBIs (this may lead to the creation of new PBIs)
- **estimation:** the team estimate the amount of work required to implement a PBI
    - **estimation metrics:**
        - **effort requires:** may be expressed in person-hours or person-days
        - **story points:** arbitrary estimate of the effort involved. story points are estimated relatively
- **creation:** new items are added to the backlog
- **prioritization:** the product backlog items are reordered

## Sprints

In Scrum, software is developed in sprints, which are fixed-length and time-boxed periods (2-4 weeks) in which software features are developed and delivered.
During a sprint the team has daily meetings (scrums) to review the progress and to update the list of work items that are incomplete.
**Sprints should produce a shippable product increment.**

**In Sprints we can distinguishh three main activities:**

1. **sprint planning:** work items to be completed in that sprint are selected and, if necessary, refined to create a **sprint backlog**. This activity should not last more than a day, at the beginning of the sprint
    1. the sprint planning allows to establish an agreed sprint goal: sprint goals may be focused on software functionality, support or performance and reliability.

2. **sprint execution:** the team work to implement the sprint backlog items that are been chosen.
3. **sprint review:** the work done in the spirnt is reviewed by the team and possibly by external stakeholders

During a scrum the sprint backlog is reviewed. Completed items are removed and new items may be added to the backlog as new information emerges.
The team decide who should work on sprint backlog items that day.
There are two practices that are strongly suggested by agile software engineers to be used in a sprint:

- **test automation:** as far as possibile product testing should be automated
- **continuous integration**

### Team Composition and Coordination

**The ideal Scrum Team size is between 5 and 8 people.** Large enough to be diverse yet small enough to communicate informally and to agree on the priorities.
The team is self-organized and has to appoint someone to take the responsabilities of product management. Usually the ScrumMaster.

The use of daily scrums as a coordination mechanism is based on 2 assumptions that are not always correct:

- scrum assumes that the team will made up of full-time workers that share a workspace
- scrum assumes that all team members can attend a moring meeting

# 3. Scenarios, Personas & Stories

**Personas** (a way of representing users) inspire **Scenarios** (natural language descriptions of a user interacting with a software product) that are developed into **Stories** (natural language descriptions of something that is needed or wanted by users), which define **Features** (fragments of product functionality)

**Product-Based software engineering needs less requirements and documentation that project-based SE because requirements are not set by customers and requirements can change.**

To identify product features (fragments of functionality) need to understand potential users.
**User Representations (Personas) and natural language descriptions (Scenarios and Stories) help identifying product features.**

## Personas

**Who are the target users for our product? Need to understand potential users to design features for them by identifying the background, skills and experience of**

**potential users.**

**Personas are (imagined) types of product users.**

Generally we need a few (1-5) personas to identify key product features

**Aspects of Personas Descriptions**

- **personalization:** you should give them a name and say something about their personal circumstances. This is important because you shouldn't think of a persona as a role but as an individual
- **job-related:** if your product target a business you should say something about their job and (if necessary) what that job involves
- **education:** you should describe their educational background and their level of technical skills
- **relevance:** if you can you should say why they might be interested in using the product

**Personas allow developers to "step into the users' shoes."**

## Scenarios

**To discover product features, we can define scenarios of user interactions with the product.**

**A Scenario is a narrative describing a situation in which a user is using our product's features to do something he wants to do.**

Scenarios facilitate communication and stimulate design creativity. Generally 3-4 scenarios for each persona are suggested.
They are written from the user's perspective and each team member should individually create some scenarios, and then discuss them with the rest of the team and with users (if possible)

## User Stories

**User Stories are finer-gran narratives that set out in a more detailed and structured way a single thing that a user wants from a software system.**
User stories should focus on a clearly defined system feature or aspect of a feature that can be implemented in a single sprint.
If a story is about more complex feature that might take several sprints is called an epic.

**You can develop user stories by refining scenarios.**

**The general template for a user story is:**

- **as a** "role"
- **I want to** "do something"

- **so that** "reason/values"
- organize and chunk work into units that represent value to the customer
    - distinguish more valuable from less important
    - visualize units of work through simple stories
- build the software form the user perspective

**Good user stories make the rest of the development process more efficient and create business value for the users.**

## Feature Identification

**Your aim in the initial stage of product design should be to create a list of features that define your product.**
A feature is a way of allowing users to access and use your product's functionality: **the feature list defines the overall functionality of the system.**

**Features should be CRI:**

- **coherent:** a feature should be linked to a single item of functionality
- **relevant:** a feature should reflect the way that users normally carry out some task
- **independent:** a feature should not depend on how other system features are implemented

## Feature Derivation

Product development team must meet to discuss scenarios and stories to extract the list of product feature descriptions.

**"Feature Name"**

- **input:** the input from the user and other sources
- **activation:** how the feature is activated by the user
- **action:** a description of how the input data are processed
- **output:** the output to the user and the system

**Start prototyping to demonstrate features.**

## Feature Creep

**Feature Creep occurs when new features are added in response to user requests without considering whether or not these features are generally useful or whether they can be implemented in some other ways.**
Too many features make products hard to use.

**There are the three reasons why feature creep occurs:**

1. product managers are reluctant to say no when users ask for specific features
2. developers try to match features in competing products
3. the product includes features to support both inexperienced and experienced users

**Avoiding Feature Creep:**

- does this feature really add anything new or is it simply an alternative way of doing something that is already supported?
- is the feature likely to be important and used by most software users?
- can this feature be implemented by extending an existing feature rather than adding a new feature to the system?
- does this feature provide general functionality or is a very specific feature?

# 4. Software Architecture

**To create good products we need to pay attention to its overall organization.**

To create a reliable, secure and efficient product you need to pay attention to architectural design, which includes:

- overall organization
- how the software is decomposed into components
- server organization
- technologies that you use

**Architecture is the fundamental organization of a software system, embodied in its components, its relationships to each other and to the environment, and the principles guiding its design and evolution.**

## Services, Components, Modules

- **service:** coherent unit of functionality
- **component:** software unit offering one or more services
- **module:** set of components

## Components

**A component is an element that implements a coherent set of functionality or features.**
Software components can be considered as a collection of one or more services that may be used by outer components: when designing the software architecture you don't have to decide how an architectural element or component is to be implemented. Rather, you design the component interface and leave the implementation of the interface to a later stage.

**Warning:** as the number of components increases, the number of relationships tends to increase at a faster rate. Simplicity is essential.

**Issues that influence the architectural decisions:**

- **non-functional product characteristics:** *optimizing one non-functional attribute affects others (e.g: security affects performance)*
    - responsiveness
    - performance
    - reliability
    - availability
    - security
    - resilience
    - ...
- **product lifetime:** if a long lifetime is in plan you need to create regual product revisions, hence need an architecture that is evolvable
- **software reuse:** using already implemented products or open-source solutions is convininent but it constrains your architecture because you must fit your design arount the software that is being reused
- **number of users**
- **software compability:** for some products it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system

## System Decompositions

**Complexity in a system architecture arises because of the number and the nature of the relationships between components in that system.**

**Design Guidelines**

- **separation of concerns:** organize your architecture into components that focus on a single concern
- **stable interfaces:** design component interfaces that are coherent and that change slowly
- **implement once:** avoid duplicating functionality at different places in your architecture

In an **layered architecture** each layer is an area of concern and is considered separately form other layers. The architectural model is a high-level model that does not include implementation information.
**Cross-Cutting Concerns are concerns that are systemic, they affect the whole system.**
In a layered architecture they affect all layers in the system as well as the way in which people use the system.

**Generally Speaking, system decomposition must be done (partly) in conjunction with choosing technologies for your system.**

## Security Architecture

Different technologies are used in different layers. Attackers can try to use vulnerabilities in these technologies to gain access.
Consequently, you need protection from attacks at each layer as well as protection at lower layers in the system from successful attacks that have occurred at higher-level layers.

If there is only a single security component in the system, this represents a critical system vulnerability. If all security checking goes through that component and it stops working properly or is compromised by an attack, then you have no reliable security in your system. By distributing security across the layers your system in more resilient to attacks and software failure.

## Distribution Architecture

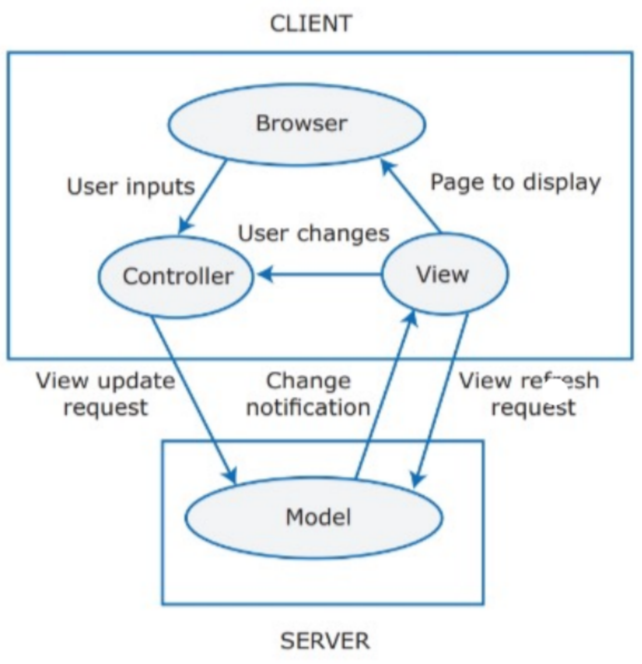
**The distribution architecture of a software system defines the servers in the system and the allocation of components to these servers.**

**Client-Server Architectures are a type of distribution architecture that is suited to applications where client access a shared database and business logic operations on that data.**
In this architecture the user interface is implemented on the user's own computer or mobile device and the functionality is distributed between the client and one or more servers.

**Model-View-Controller Pattern:**

- architectural pattern for client-server interaction
- model is decoupled from its presentation

- each view registers with model so that if model changes all views can be update



## Service-Oriented Architecture

**Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another.**

A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

**To choose a distribution architecture we must consider**

- **data type and updates:** if you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provide locking and transaction management
- **change frequency:** if you anticipate that system components will be regularly changed or replaced, then isolating these components as a separate services simplifies those changes.
- **the system execution platform:** if you plan to run your system on the cloud with users accessing it over the internet it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. If your product is a business system that runs on local servers then a multi-tier architecture may be more appropriate

# Technology Choices

**The technology choices must concentrate on components like databases, platform, server, open source software and development tools. These choices affect and constraint the overall system architecture and it's difficult to change them during the development phase.**

## Database

Two kinds of DBs:

- **relational:** suitable for situations where you need transaction management and data structures are predictable
- **non-relational:** more flexible, data organized hierarchly rather than tables

## Delivery Platform

**delivery can be as a web-based or a mobile product, or both.**
Mobile platforms have more issues:

- intermittent connectivity
- lack of processing power
- battery life

## Server

**A key decision that you have to make is whether to design you system to run on customer servers or run on the cloud.**
Some business are concerned about cloud security and prefer to run their systems on in-house servers.
An important choice you have to make if you are running your software on the cloud is which cloud provider to use and to avoid vendor lock-in.

## Development Technology

Development technology influence the architecture of your product, e.g, many web development frameworks assume use of the MVC architectural pattern.
The development technology that you use may indirectly influence the architecture of your product.

# EIP - Enterprise Integration Patters

**Enterprise Application:**

- **various heterogeneous services**
  - sources and sinks
  - some you are in control of, others not
- **various heterogeneous data types**
  - different representations, even for analogous data
- **multiple different participants**
  - different organizations agreeing to share data
- **all interconnected via network**

**Enterprise applications are complex distributed multi-service applications whose services must play together, being them suitably integrated.**

The architectural question is how to integrate multiple different services to realize enterprise applications that are:

- **coherent**
- **extensible**
- **maintainable**
- **simple to understand**
- Enterprise Integration Integration allows to manage the complexity behind every system, tolerate the changes and to be unified model by introducing pattern based methods.

## Patterns

**A pattern is an high-level abstraction of accepted, reusable solution to recurring problem.**
**When facing a problem, considering existing patterns that are applicable to solve such problem saves us from re-inventing the wheel and making the same mistakes as others.**

Patterns are given in terms of the **following schema:**

- problem statement
- context
- forces
- solution
- **enterprise applications are big**, comprising legacy and commercial software components, and owned and third-party services
- **integration is the process for making various eternogeneous bits of software work together**, seamlessy to fulfill some buisness workflows
- **an EIP is a reusable abstraction of proven solutions to well-known problems raising while integrating the software components/services forming enterprise applications**

## Messages

**All systems need to communicate with each other: internally the same happens for components and subsystems. This evolves around a message, a discrete piece of information sent from a service to another.**

A message is typically structured into header (considered as metadata) and body (the real data payload).

**We can distinguish three types of messages:**

- **document message:** pure data
- **event message:** something has happened
- **command message:** do something

The message-based communication allows to create services that are loosely coupled by using a simple exchange pattern (a one-way pattern between sender and receiver).
The communication is realized via **channels**, abstractions for components sending messages.
Under the hood they can be implemented in many ways (RPC, HTTPS, ...).
**Channels are one-way so communications are natively asynchronous**: to realize a synchronous communication we need to use different ("parallel") channels.

**We can also distinguish in different types of channels:**

- **point-to-point**
- **publish-subscribe**

Application services are typically independent of the messaging systems:

- **adapters** enable application-specific data to be sent to channels
- **message endpoints** enable application services to send/receive messages to/from channels

**The simplest integration:** Message endpoints and channels enable realizing the simplest integration possible:

- message endpoints for services to send/receive messages
- channels to transport messages from a service to another

Sending and receiving services may expect different data types or formats from both the header and the body of the message that your application use.
We introduce a **message translator** to be able to decode the correct/expected message format.
**Translators enable transforming messages.**

## Pipes and Filters

Beside the translators other steps may be needed, for instance we need to reason about:

- how to route messages?
- how to split messages?
- how to aggregate messages?

**The pipe and filters architectural style (plus other EIPs) enable structuring the more complex integration needed by enterprise applications: messages pass through multiple steps/components processing it (the filters). These components send the message down the channels (pipes) they are connected to. Note that pipes and filters all deal with the same message/channel abstraction, and can be composed flexibly depending on the circumstances.**

# 5. Cloud-Based Software

The cloud is made up of very large number of remote servers that are offered for rent by companies that own these servers.
**Cloud-Based Servers are virtual servers**, which means they are implemented in software.
You can rent as many servers as you need, run your software on these server and make them available to your customers: cloud servers can be started up and shut down as demand changes.

**Cloud Advantages: CRES**

- **scalability:** reflects the ability of your software to cope with increasing numbers of users
- **elasticity:** is related to scalability but also allows for scaling down as well as scaling up
- **resilience:** resilience: means that you can design your software to tolerate server failures
- **cost:** avoid the initial capital costs of hardware, shifting to a pay-per-use paradigm

## Virtualization: From VMs to Containers

**Containers are an operating system virtualization technology that allows indipendent servers to share a single operating system.**
They are particularly useful for providing isolated application services where each user sees their own version of an application.

### Docker

**Docker** is one of the most known containerization technology in the world: **Docker is a platform that allows us to run applications in an isolated environment. Docker allows us to develop and run portable applications by exploiting containers.**

Docker is a container management system that allows user to define the software to be include in a container as a docker image.
It also includes a run-time system that can create and manage containers using these Docker images.
**Docker Container System is composed by:**

- **docker deamon:** a process that runs on the host server and is used to setup, start, stop and monitor containers, as well as building and managing local images.
- **docker client:** software used by developers and system managers to define and control containers
- **dockerfile:** define the runnable application (image) as a series of setup commands that specify the software to be included in the container. Each container is defined by an associated dockerfille
- **image:** a dockerfile is interpreted to create a docker image, which is a set of directories with the specified software and data installed in the right places. Images are set up to be runnable docker applications.
  - **docker images are directories that can be archived, shared and run on different docker hosts. Everything that's needed to run a software system is included in the directory.**
  - images are read-only template to create containers
  - a docker image is a base layer usually taken from the docker registry, with your own software and data as a layer on top of this: the layered model means that updating docker applications is fast and efficient
  - each update to the filesystem is a layer on top of the existing system
- **docker hub:** registry of images that has been created. These may be reused to setup containers or as a starting point for defining new images
- **containers: containers are executing images. An image is loaded into a container and the application defined by the image starts execution.** Containers may be moved from server to server without modification and replicated across many server. You can make changes to a docker container (by modifying its files) but then you must commit these changes to create a new image and restart the container

Remember that containers are **ephimeral**, to proivde persistency we need to use volumnes that can
be mounted to running containers.

**Containers solve the problem of software dependencies. You dont have to worry about the libraries and other software on the application server being different from thos on your development server. Instead of shipping your product as a stand-alone software, you can ship a container that includes all the support software that your product needs.**
They provide a mecanism for software portability across different clouds.
They provide an efficient mechanism for implementing software services and so support the development of service-oriented architectures.
They simplify the adoption of DevOps.

**Compose is a tool for defining and running multi-container docker applications.**
With compose you can use a YAML file to configure your application services, then with a single command you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows.
It also has commands for managing the whole lifecycle of your application:

- start, stop and rebuild services
- view the status of running services
- stream the log output of running services
- run a one-off command on a service

## Everything as a Service

The idea of a service that is rented rather than owned is fundamental to cloud computing:

- **IaaS:** cloud provides offer different kinds of infrastructure
- **PaaS:** provider offers both infrastructure and platform (OS, libraries, ...)
- **SaaS:** your software runs on the cloud and is accessed through web browser or mobile app.

### SaaS: Software as a Service

**Allows to deliver product as a service in which customers do not have to install software, they pay a subscription and access product from remote.**

**Benefits:**

- **cash flow**
- **update management**
- **continuos deployment**
- **payment flexibility**
- **try before you buy**
- **data collection**

**Cons for the Customers**

- **privacy regulatio conformance**
- **security concerns**
- **network contraints**
- ...

**Some issues when desining for a SaaS product:**

- **local vs remote processing:** some features can be executed only locally, this can reduce network traffic, increasing response and speed, but can slow increase consumption for battery-powered devices
- **authentication**

- **information leakage**
- **multi-tenant vs multi-instance database management:** single repository vs separate copies of system and database

## Multi-Tenant Systems

**A multi-tenant database is partitioned so that customer have their own space and can store and access their own data. There is a single database schema, defined by the SaaS provider, that is shared by all of the system's users.**
Items in the database are tagged with a tenant identifier, representing a company that has stored data in the system. The database access software uses this tenant identifier to provide "logical isolation", which means that users seem to be working with their own database.

Mid-size and large businesses rarely want to use generic multi-tenant software, often prefer a customized version adapted to their own requirements, by customizing:

- **authentication:** businesses may want users to authenticate using their business credentials rather that the account credentials set up by the software provider
- **branding:** businesses may want a user interface that is branded to reflect their own organization
  - **UI configurability** can be implemented by employing user profiles: users are asked to select their organization or provide their business email address. Product uses profile information to create a personalized version of the interface
- **business rules:** businesses may want to be able to define their own business rules and workflows that apply to their own data
- **data schemas:** businesses may want to be able to extend the standard data model used in the system database to meet their own business needs
- **access control:** business may want to be able to define their own access control model that sets out the data that specific users or user groups can access and the allowed operations on that data

**To adapt the DB schema** in a multi-tenant system we can have two solutions:

1. you add some extra columns to each database table and define a customer profile that maps the column names that the customer wants to these extra columns. However it is difficult to know how many extra columns you should include
2. allow customers to add any number of additional fields and to define the names, types and values of these fields. The names and types of these values are held in a separate table, accessed using the tenant identifier. Unfortunately, using tables this way adds a complexity to the DBMS

Information from all customers is stored in the same database in a multi-tenant system so a software bug or an attack could lead to the data of some or all customers being exposed to

others.
The **key security issues** are:

- **multilevel access control** means that access to data must be controlled at both the organizational level and the individual level. You need to have organizational level access control to ensure that any database operation only act on the organization's data. The individual user accessing the data should also have their own access permissions
- **encryption:** in a multitenant database reassures corporate users that their data cannot be viewed by people from other companies if some kind of system failure occurs

## Multi-Instance Systems

**Multi-Instance Systems are SaaS systems where each customer has its own system that is adapted to his needs, including its own database and security controls. Multi-instance, cloud based systems are conceptually simpler that multi-tenant systems and avoid security concerns such as data leakage from one organization to another.**

There are **two types** of multi-instance system:

- **VM-Based:** the software instance and database for each customer runs in its own virtual machine. All users from the same customer may access the shared system database
- **Container-Based** each user has an isolated version of the software and database running in a set of containers

**Pros of Multi-Instance Systems:**

- **flexibility:** each instance of the software can be tailored and adapted to a customer's needs
- **security:** each customer has their own database so there is no possibility of data leakage
- **scalability:** instances of the system can be scaled according to the needs of individual customers
- **resilience:** if a software failure occurs, this will probably only affect a single customers

**Cons of Multi-Instance Systems:**

- **cost:** it is more expensive to use multi-instance systems because the costs of renting many VMs in the cloud is higher and the costs of managing multiple systems can be high
- **update management:** it is more complex to manage updates to the software because many instances have to be updated

## Scalability

**The scalability of a system reflects its ability to adapt automatically to changes in the load on that system.**
**Scalability is achieved through making possible to add new virtual servers (scaling out) or to increase the power of a system server (scaling up) in response to increasing load.**

In cloud-based systems, scaling-out rather than scaling-up is the normal approach used. Your software has to be organized so that individual software components can be replicated and run in parallel

## Resilience

**The resilience of a system reflects its ability to continue to deliver critical services in the event of a system failure or a malicious system use.**
**To achieve resilience you need to be able to restart your software quickly after an hw/sw failure**

**Resilience relies on redundancy:**

- **hot standby:** replicas of the software and data are maintained in different locations, database updates are mirrored and a system monitor continually checks the system status and can switch to the standby system automatically
- **cool standby:** redundant virtual servers, not hosted in the same physical computer and ideally in a different datacenter. If a physical server fails or if there is a wider datacenter failure the operation are automatically switched to the software copies elsewhere

## Architectural Decisions

There are **three possible ways of providing a customer database in a cloud-based system:**

- as a **multi-tenant system**, shared by all customers for your product
- as a **multi-instance system** with each customer database running on **its own VM**
- as a **multi-instance system** with each database running on **its own container**. The customer database may be distributed over several containers

**The factors to observe regarding the choice are:**

- **target customer:** do customers require different database schemas and database personalization? If so, use a multi-instance DB
- **transaction requirements:** it is critical that your product support ACID transactions where the data is guaranteed to be consistent at all times? Use a multi-tenant database or a VM-based multi-instance database
- **database size and connectivity:** how large is the typical database used by customers? A multi-tenant model is usually best for very large databases as you can

focus effort on optimizing performance
- **database interoperability:** will customers wish to transfer information from existing DBs? What are the different schemas? If customers have many different schemas a multi-instance DB should be used
- **system structure:** are you using a service-oriented architecture for your system? Can customer databases be split into a set of individual service databases? If so use containerized multi-instance databases

**Different types of customers have different expectations about software products:**

- **small customers:** do not expect branding and personalization: can use a multi-tenant database with a single schema
- **large companies:** possible to some extent to use a multi-tenant, easier with a multi-instance
    - in products in which the database has to be consistent at all times (e.g.: finance) you need a transaction-based system: either a multi-tenant or a database per customer running on a VM

## Software Structure

- **monolithic approach:** distribution across multiple servers running large software components. The traditional multi-tier client-server architecture is based on this distributed system model
- **service-oriented approach:** the system is decomposed into fine-grain stateless services. Since it is stateless each service is independent and can be replicated, distributed and migrated from one server to another. The service oriented approach is particularly suitable for cloud-based software with services deployed in containers.

# Kubernetes

**Kubernetes is a container orchestration platform**

## Design Principles: D.D.D.I.S

- **declarative:** We define the desired state: K8s will detect when the actual state do not meet the desired state and will intervene to fix the problem
- **distribution:** K8s provide a unified interface for interacting with a cluster of machines
- **decoupling:** K8s naturally supports the idea of decoupled services which can be scaled and updated indipendently
- **immutable structure:** build a new container image, deploy and terminate the old version instead of modifying the same container
    - makes it easier to rollback

## K8s Desired State

Defined by a collection of objects. Each object has a specification that specify the desired state of the object and a status, which reflects the current state of the object.
K8s objects are defined in manifests (yaml or json files)

### K8s Objects

- **pods:** one or more (tightly) related containers. Pods have a shared network layer and a shared file system volumes
- **deployments:** include a collection of pods defined by a template and a replica count (number n of how many copies of the template we want to run)
- **service:** each pod is assigned a unique ip address to communicate with it. Service provides a stable endpoint to direct traffic to the desired pods
- **ingress:** to expose over application to traffic external to our cluster we define an ingress object

## K8's Control Plane

**The control plane components make global decisions about the cluster (e.g.: scheduling), as well as detecting and responding to cluster events (e.g.: starting up a new pod)**

There are 2 types of machines in a cluster:

- **master node:** (often a single) machine that contains most of the control plane components
- **worker nodes:** machines that runs the app's workload

**Components of the Master Node:**

- **api-server:** is the frontend for the control plane. Validates user requests and act as a unified interface for questions about the cluster state
  - the cluster state is stored in the "etcd", a <k,v> database
- **scheduler:** determines where object should run
  - asks the api-server which objects havent been assigned a machine
  - determines which machine use
  - give the assigment to the api-server
- **control-manager:** monitor cluster state trough the api-server. If the state differs from the desired state it will take action (through the api-server)

**Components of the Worker Node:**
Node components run on every node, mantaining running pods and providing the kubernetes runtime env:

- **kublet:** an agent that runs on each node (a virtual/physical machine) in the cluster. It makes sure that containers are running in a pod

- **kube-proxy:** network proxy that runs in every node of the cluster, implementing part of the service concepts. Kube-proxy maintain network rules on nodes

## Concluding Remarks

**Don't use K8s:**

- if you can run your workload on a single machine
- if your compute needs are light
- if don't need high availability and/or can tolerate downtime
- don't envision lots of changes to your deployed services
- you have a monolith and dont plan to break it into microservices

# 6. Microservices

**A software service is a software component that can be accessed from remote computers over the internet.**
Given an input, a service produces a corresponding output without side effects: the service is accessed through its published interface and all details of the service implementation are hidden.
Usually services do not maintain internal state. State information is either stored in a database or is maintained by the service requestor.

When a service request is made, the state information may be included as part of the request and updated state information is returned as part of the service result. As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers.

**Microservices are small-scale, stateless, services that have a single responsibility**

**Let's define every term of our microservice informal definition:**

- **self-contained:** microservices do not have external dependencies. They manage their own data and implement their own user interface
- **lightweight:** microservices communicate using lightweight protocols, so that service communication overheads are low
- **implementation-independent:** microservices may be implemented using different programming languages and may use different technologies in their implementation
- **independently deployable:** each microservice runs in its own process and is independently deployable, using automated systems
- **business-oriented:** microservices should implement business capabilities and needs, rather than simply provide a technical service

**A well-designed microservice should have high cohesion and low coupling**

- **cohesion:** measure of the number of relationships that parts of a component have with each other. High cohesion means that all the parts that are needed to deliver the component's functionality are included in the component
- **coupling:** is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components

**Each microservice should have a single responsibility i.e. it should do one thing only and it should it well.**
However "one thing only" is difficult to define in a way that's applicable to all services: responsibility does not always mean a single, functional activity.

**How big should a microservice be?**
The general rule is called "rule of twos": service can be developed, tested, and deployed by a team in two weeks. Team can be fed with two large pizzas (8-10 people)

## Microservices Architecture

**A microservices architecture is an architectural style, a tried and tested way of implementing a logical software architecture.**
This architectural style address two problems with monolithic applications:

- **problem 1:** the whole system has to be rebuilt, re-tested and re-deployed when any change is made. this can be a slow process as changes to one part of the system can adversely affect other components
- **problem 2:** as the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement most popular system functions

**Microservices are self-contained and run in separate processes. In cloud-based systems, each microservice may be deployed in its own container. This means a microservice can be stopped and restarted without affecting other parts of the system.**
If the demand on a service increases, service replicas can be quickly created and deployed. These do not require a more powerful server so "scaling-out" is typically much cheaper than "scaling-up"

### Service Communications

Services communicate by exchanging messages that include information about the originator of the massage, as well as the data that is the input to or output from the request. When you are desining a microservices architecture, you have to establish a standard for communications that all microservices should follow.
There are two types of interaction: **synchronous** (direct) and **asynchronous** (indirect).

## Data Distribution and Sharing

**You should isolate data within each system service with as little data sharing as possible: if data sharing is unavoidable, you should design microservices so that most sharing is read-only, with a minimal number of services for data updates.**

If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.
**Microservices systems must be designed to tolerate some degree of data inconsistency.** Two types of inconsistency have to be managed:

- **dependent data inconsistency:** actions/failures of one service can cause data managed by another service to become inconsistent
- **replica inconsistency:** several replicas of the same service may be executing concurrently. Each with its own DB copy, each updates its own db copy, so there is a need to make these dbs "eventually consistent"

A **fundamental theorem in distributed system** is the **CAP Theorem (Consistency, Availability, Partition Tolerance):**
*In presence of a network Partition, you cannot have both Availability and Consistency*

- **consistency:** any read operation that begins after a write operation must return that value, or the result of a later write operation
- **availability:** every request received from a non-failing node must result in a response
- **network partition:** network can lose arbitrarily many messages sent from one group to another

## Failure Management

**Services must be designed to cope with failures like:**

- **internal service failure:** these are conditions that are detected by the service and can be reported to the service client in an error message
- **external service failure:** these failures have an external cause, which effects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken
- **service performance failure:** the performance of the service degrades to an unacceptable level

**Some mechanism used to manage failures are timeouts and circuit braker**

- **timeouts:** after a given time the calling service assumes that the service requests has failed and acts accordingly. The problem is that waiting the timeout slows the whole system

- **circuit brakers:** like an electrical circuit braker this immediately denies access to a failed service without the delays associated with timeouts

## REST

**The REST (Representational State Transfer) architectural style is based on the idea of transferring representations of digital resources from a server to a client.**
Resources are accessed via their unique URI and RESTful services operate on these resources.

**REST follow some simple REST principles:**

- **resource identification through URIs:** service exposes set of resources identified by URIs
- **uniform interface:** clients invoke HTTP methods to create/read/update/delete resources
- **self-descriptive messages:**
    - requests contain enough context information to process message
    - resources decoupled from their representation so that content can be accessed in a variety of formats
- **stateful interactions through hyperlinks**
    - every interaction with a resource is stateless
    - server contains no client state, any session state is held on the client
    - stateful interactions rely on the concept of explicit state transfer

## Service Deployment Automation

The service development team decide which programming language, database, libraries and other support software should be used to implement their service.
Consequently, there is no standard deployment configuration for all services.
It is now normal practice for microservice dev teams to be responsible for deployment and service management as well as software development and to use **continuous development**, as soon as a change to a service has been made and validated, the modified service is redeployed.

Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software. if the software passes these tests then it enters another automated pipeline that packages and deploy the software.

Testing cannot prevent 100% of unanticipated problems so there is a need to monitor the deployed services. Monitoring allows to roll back to previous version in case of service failure.

## Architectural Smells and Refactorings

**An architectural smell is a commonly used architectural decisions that negatively impacts system lifecycle qualities.**

The **design principles** to follow to avoid architectural smells D.ID.IOF.HS:

- **independent deployability:** the microservices forming an application should be independently deployable
- **horizontal scalability:** the microservices forming an application should be horizontally scalable providing the possibility of adding/removing replicas of single microservices
- **isolation of failures:** failures should be isolated
- **decentralization:** decentralization should occur in all aspects of microservice-based applications, from data management to governance

## MicroFreshener

**MicroFreshener is a tool for editing app specifications, automatically identifying architectural smells and applying architectural refactoring to resolve the identified smells**

**Smells relevated and proposed refactoring of MicroFreshener:**

- **no api gateaway:** the no api-gateway smell occurs when the external clients of an application directly interact with some internal services. If one of such services is scaled out, the horizontal scalability of microservices may get violated because external clients may keep invoking the same instance. The **proposed solution is to add an api gateway message router (MR)**
- **shared persistency:** the shared persistency smell occurs whenever multiple services access or manage the same DB, possibly violating the decentralization design principle. The **proposed solution is to add a data manager**. This adds the complexity of a new microservice to implement and maintain. **A smarter refactoring consists of using stats as data manager**
- **wobbly service interaction:** the interaction of microservices is said wobbly when a failure of a service can trigger a failure in another service(s). **The proposed solution is to use timeouts.**
- **endpoint-based interaction:** the endpoint based interaction smell occurs when one or more of its microservices invoke a specific instance of another microservice e.g., because its location is hardcoded in the source code of the invoker. If this is the case when scaling out the invoked service the replicas are not reached. **The proposed solution is to add a service discovery**

# 7. Security and Privacy

**Software security should always be a high priority for product developers and their users: if you do not prioritize security you and your customers will inevitably suffer**

**losses from malicious attacks.**

Three types of security threats:

- **availability threats:** an attacker attempts to deny access to the system for legitimate users
- **integrity threats:** an attacker attempts to damage the system or its data
- **confidentiality threats:** an attacker tries to gain access to private information held by the system

Security is a system wide-issue: application software depends on operating system, web server, language run-time system, etc. Attacks may target any level of system infrastructure stack, starting from the network.

**System management activities to maintain security:**

- **authentication and authorization standards** and procedures to ensure that all users have strong authentication and properly set up access permissions
- **system infrastructure management** to keep infrastructure software properly configured and to promptly apply security updates patching vulnerabilities
- **regularly monitoring attacks** to promptly detect them and trigger resistance strategies to minimize effects of attack
- **backup policies** to keep undamaged copies of program and data files that can be restored after an attack

## Attacks and Defenses

### Injection Attacks

Malicious user uses a valid input field to input malicious code or database commands to damage the system. Common type of injection attacks include buffer overflow and sql poisioning

### Session Hijacking

A session is a period of time during which user's authentication with a web app is valid. Generally the session cookie is sent from server to client which sends the token in each HTTP request so user does not have to re-authenticate for subsequent system interactions. The session is closed when the user logs out or when system times out. In this attack the attacker acquires a valid session cookie and impersonates a legitimate user.

Two types of hijacking:

- **passive:** attacker monitors client-server traffic looking for valuable informatio
- **active:** attacker carries out user actions on server

The **defenses** adopted can be to encrypt client-server network traffic via HTTPS or use multi-factor authentication to require confermation of new actions that may be damaging.

### Cross-Site Scripting Attack

An attacker adds malicious javascript code to the web page in the user's browser. The malicious script may steal customer information or direct them to another website.

The **defenses** are represented by input validation, check input from database before adding it to generated page and employ the html "encode" command

### Denial of Service

Attacks that are intended to make system unavailable for normal use. The historical way to carry out a DoS attack was by exploiting the TCP 3-way handshake by exhausting server resources.
Nowadays it is generally called Distributed DoS which involve distributed computers that have usually been hijacked to form a botnet sending hundreds of thousands of requests.

Some **defenses** can be represented by temporary user lockouts and IP address tracking and blacklist them.

### Brute Force

Attacks on web application where the attacker has some information, such as a valid login name, but does not have the password for the name. The attacker creates different passwords and tries to login with each of them.

Possible **defenses** are to convince or force users to set long strong password that are not in a dictionary and are not common words, or use two-factors authentication.

## Authentication

**Authentication is the process of ensuring that a user of your system is who they claim to be.**
You need authentication in all software products that maintain user information, so that only the providers of that information can access and change it.
You also use authentication to learn about your users so that you can personalize their experience of using you product.

There are many approach to authenticate an users:

- **knowledge-based:** relies on providing a secret
    - there are many weakness:
        - users choose insecure passwords that are easy to remember
        - phishing

    - same password for many sites
    - recovery password systems may be vulnerable
  - defenses are to force users to use strong passwords or to add further knowledge (secret questions)
- **possession-based:** relies on having a physical device
- **attribute-based:** relies on a unique biometric attribute of the user

### Federated Identity

**Federated identity is an approach to authentication where you use an external authentication service such as "Login with Google".**

The advantage for a user is that they have a single set of credentials that are stored by a trusted identify service.
The main drawback is that the product provider must share information with external services.

Another useful method is the mobile device authentication in which we install an authentication token generator on mobile devices.

## Authorization

**Authorization is a complementary process in which that identity is used to control access to software system resources.**
This policy is a set of rules that define what information (data and programs) is controlled, who has access to that information and the type of access that is controlled: access control policy must reflect data protection rules that limit access to personal data to prevent legal actions in case of data breach.

ACLs are widely used to implement access control policies:

- classify individuals intro groups dramatically reduce ACLs size
- different groups can have different rights on different resources
- hierarchies of groups allow to assign rights to subgroups/individuals

## Encryption

**Encryption is the process of making a document unreadable by applying an algorithmic transformation to it.**
A secret key is used by the encryption algorithm as the basis of this transformation.
You can decode the encrypted text by applying the reverse transformation.

- **symmetric encryption:** one secret key both for encryption and decryption
- **asymmetric encryption:** each user has a public and a secret key

- the https protocol is a standard protocol for securely exchanging texts on the web. It is the standard http protocol plus an encryption layer called TLS (transport layer security). TLS is used for two things:
    - verify the identity of the web server
    - encrypt communications so that they cannot be read by an attacker who intercepts the messages between the client and the server

**Data encryption can be used to reduce the damage that may occur from data theft.** If information is encrypted it is impossible for thieves to access and use the unencrypted data. You should encrypt user data whenever it is practicable to do so:

- **data in transit:** when transferring the data over the internet you should always use the https rather than the http protocol to ensure encryption
- **data in rest:** if data is not being used then the files where the data is stored should be encrypted so that theft of these files will not lead to disclosure of confidential information
- **data in use:** the data is being actively processed. Encrypting and decrypting the data slows down the system response time. Implementing a general search mechanism with encrypted data is impossible

## Key Management

**Key management is the process of ensuring that encryption keys are securely generated, stored and accessed by authorized users.**
It is impractical to do key management manually and you need to use some kind of automated key management system (KMS)

## Privacy

**Privacy is the social concept that relates to the collection, dissemination and appropriate use of personal information held by a third-party such as a company or a hospital.**

**General principles to follow** when manage privacy are:

- **awareness and control:** users of your product must be made aware of what data is collected when they are using your product, and must have control over the personal information that you collect from them
- **purpose:** you must tell users why data is being collected and you must not use that data for other purposes
- **consent:** you must always have the consent of a user before you disclose their data to other people
- **data lifetime:** you must not keep data for longer than you need to. If a user deletes their account, you must delete the personal data associated with that account

- **secure storage:** you must maintain data securely so that it cannot be tampered with or disclosed to unauthorized people
- **discover and error correction:** you must allow users to find out what personal data that you store. You must provide a way for users to correct errors in their personal data
- **location:** you must not store data in countries where weaker data protection laws apply unless there is an explicit agreement that the stronger data protection rules will be upheld

## Security and Microservices

**Discussing security in microservices, we need to make a comparison respect to monolith approach to security.**
**In a monolith architecture the inter-component communication happen inside a single process, despite in microservices and broader attack surface represent an higher risk because the inter-service communication happen via remote calls so there is a large number of entry points.**

In a monolith architecture a security screening it's execute one and request are dispatched to corresponding component.
Differently in microservices each microservice has to carry out independent security screening that may need to connect to a remote security token service.
The repeated distributed security checks surely also have an impact on services performance. As work-around we can use the notion of **trust-the-network** for which we skip security checks at each (internal) microservices: this is a different philosophy compared to the industry trends towards a **zero trust** networking principles.
Service-to-service communication must take place on protected channels: for example, if we use certificates, each microservice must be provisioned with a certificate and a corresponding private key to authenticate itself to another microservice during interactions. The recipient microservice must know how to validate the certificate associated with calling microservice: there is a way to bootstrap trust between microservices by introducing **automation** for passing secrets to a large-scale deployments of hundreds of microservices.

Another point is represented by tracing requests: we define a log as the recording of an event of service. Logs can be aggregated to produce metrics that reflects the system state and that may trigger alert. Traces help track a request from point where it enters the system to the point where it leaves the system.
Differently from monolithic applications, a request to a microservices deployment may span multiple microservices so **correlating request among microservices is challenging.**

As mentioned, **containers are immutable server** so they don't change state after spin up but for each service we need to maintain a dynamic list of allowed clients and dynamic set of access control policies, so each service must also maintains its own credentials which need to be rotated periodically.
In a distributed environment the sharing of user context is harder than in a monolithic

architecture: we need to build trust between microservices to that receiving microservice accepts user context passed from the calling microservice.
A popular solution is to use **JSON Web Token** to share user context among microservices: the key idea is that the message carrying attributes in a cryptographically safe way.

**To Summarize:**

- the broader the attack surface, the higher the risk
- distributed security screening affects performance
- bootstrapping trust among microservices needs automation
- tracing requests spanning multiple microservices is challenging
- distribution makes sharing user context harder
- security responsibilities distributed among different teams

## Smells and Refactoring for Microservices Security

10 smells associated with 10 refactor solution, potentially affect the three security properties:

- **confidentiality:** degree to which a product or a system ensures that data are accessible only to those authorized to have access
- **integrity:** degree to which a system, product, or component prevents unauthorized modification of computer programs or data
- **authenticity:** degree to which the identity of a subject or resource can be proved to be the one claimed

**Here we analyze some potential properties violation by identifying a smell and propose a refactoring:**

- **smell:** insufficient access control
  - **property affected:** confidentiality
  - **refactoring:** use OAuth 2.0 by
- **smell:** publicly accessible microservice
  - **property affected:** confidentiality
  - **refactoring:** add API gateway
- **smell:** unnecessary privileges to microservices
  - **properties affected:** confidentiality, integrity
  - **refactoring:** follow the least privilege principle
- **smell:** "home-made" crypto code
  - **properties affected:** confidentiality, integrity, authenticity
  - **refactoring:** use established encryption techniques
- **smell:** non-encrypted data exposure
  - **properties affected:** confidentiality, integrity, authenticity
  - **refactoring:** encrypt all sensitive data at rest

- **smell:** hardcoded secrets
  - **properties affected:** confidentiality, integrity, authenticity
  - **refactoring:** encrypt secrets at rest
- **smell:** non-secured service-to-service communications
  - **properties affected:** confidentiality, integrity, authenticity
  - **refactoring:** use mutual TLS
- **smell:** unauthenticated traffic
  - **properties affected:** authenticity
  - **refactoring:** use mutual TLS and use OpenID conncet
- ...

## OWASP API Security

Complexity of source code leads to more security vulnerabilities so exponential increase of defects as number of lines of code increases.

**Top 10 API Vulnerabilities:**

1. **broken object-level authorization:** attacker substitutes ID of their resource in API call with an ID of a resource belonging to another user. Lack of proper authorization checks allows access. This attack is known as IDOR (insecure direct object reference).
   1. how to prevent:
      1. implement authorization checks with user policies and hierarchy
      2. check authorization each time there is a client request to access database
      3. ...
2. **broken authentication:** poorly implemented API authentication allowing attackers to assume other users' identities
   1. how to prevent:
      1. check all possible ways to authenticate to all APIs password reset APIs and one-time links also allow users to get authenticated and should be protected just as seriously
      2. use standard authentication, token, password
      3. use short-lived access tokens
3. **excessive data exposure:** API exposing a lot more data than the client legitimately needs, relying on the client to do the filtering. Attacker goes directly to the API and has it all
   1. how to prevent:
      1. never rely on client to filter data
      2. define schemas of all the API responses
4. **lock of resources & rate limiting:** API is not protected against an excessive amount of calls or payload sizes. Attackers use that for DoS and brute force attacks
   1. how to prevent:

        1. rate limiting
        2. payload size limits
        3. ...
5. **broken function level authorization:** API relies on client to use user level or admin level APIs. Attacker figures out "hidden" admin API methods and invokes them directly
    1. how to prevent:
        1. don't rely on app to enforce admin access
        2. properly design and test authorization
6. **security misconfiguration:** poor configuration of the API servers allows attackers to exploit them
    1. how to prevent:
        1. disable unnecessary features
        2. automate process to locate configuration flaws
        3. repeatable hardening and patching processes
7. **injection:** attacker constructs API calls that include SQL, NoSQL, LDAP and other commands that the API or backend behind it blindly executes
    1. how to prevent:
        1. never trust your API consumers, even if internal
        2. validate, filter, sanitize all incoming data
        3. ...
8. **improper assets management:** attacker finds non-production version of the API
    1. how to prevent:
        1. inventory all API hosts
        2. limit access to anything that should not be public
        3. additional external controls such as API firewalls
9. **insufficient logging and monitoring:** attacks go unnoticed
    1. how to prevent:
        1. ensure that logs are formatted to be consumable by other tools
        2. protect logs as highly sensitive
        3. include enough detail to identify attackers
        4. ...

# 8. DevOps and Code Management

## DevOps

**Traditionally separate teams were responsible of software development, software release and software support.**
**To speed up the release and support processes, an alternative is called DevOps or Development + Operations.**
**DevOps integrates development, deployment and support in a single team.**

Three factors enabling this paradigm shift:

- agile software engineering reduced software development time: traditional release process became a bottleneck
- amazon re-engineered its software into (micro)services, assigning both service development and service support to the same team
- SaaS release of software became possible

**The DevOps philosophy:**

- **everyone is responsible for everything:** all team members have joint responsibility for developing, delivering and supporting the software
- **everything that can be automated should be automated:** all/most activities involved in testing, deployment and support should be automated
- **measure first, change later:** devops should be driven by measuring collected data about the system and its operation

**The DevOps benefits:**

- **faster deployment:** dramatic reduction of human communication delays, hence faster deployment to production
- **reduced risk:** small functionality increment in each release, hence less chance of feature interactions and system failures
- **faster repair:** no need to discover which team is responsible for fixing problem and to wait for them to fix it
- **more productive teams:** devops teams more productive than teams involved in separate activities

**DevOps culture means creating a team bringing together a number of different skillset.**
To create a successful team should be present a culture of mutual respect and sharing in which everyone on the team should be involved in scums and other team meetings.

## DevOps Measurement

**You should try to continuously improve your devops process to achieve faster deployment of better quality software: this implies to measure and analyze product and process data.**

Different **types of measures:**

- **process measurements:** to collect & analyze data on development/testing/deployment process
- **service measurements:** to collect & analyze data on software's performance/reliability/acceptability to customers

- **usage measurements:** to collect & analyze data on how customers use your product
- **business success measurements:** to collect & analyze data on how your product contributes to overall business success

Measuring software and its development is a complex process because need to identify suitable metrics that give you useful insights that need to be inferred from other metrics.
Also, software measurement should be automated as far as possible.

## Code Management

**Code management is a set of software-supported practices that is used to manage an evolving codebase.**
**You need code management to ensure that changes made by different developers do not interfere with each other, and to create different product versions.**
Code management tools make it easy to create an executable product form its source code files and to run automated tests on that product.

**The main objective of code management is to manage evolving project codebase to allow different versions of components and entire system to be stored and retrieved so developers can work in parallel without interfering with each other and integrate their work**

- **code transfer:** developers download code into personal file store, work on it, return it to shared code management system
- **version storage and retrieval:** files may be stored in several different versions, specific versions can be retrieved
- **branching and merging:** parallel development branches can be created for concurrent working, Changes made in different branches may be merged
- **version information:** information on different versions may be stored and retrieved

A **source code management system** is usually formed by a shared repository so all source code files and file versions are stored in the repository, together with other artifacts (conf files, libraries, ...)
The repository includes a database of information about the stored files.
Files can be transferred to/from repository, information about different versions of files and their relationships can be updated.

**Main features:**

- **version and release identification:** managed versions of a code file are uniquely identified when submitted to the system so managed files can never be overwritten
- **change history recording:** when a change to a code file is submitted, the submitter must add a string explaining the reasons of the change
- **independent development:** several developers can work on the same code file at the same time. When this is submitted to the code management system a new version is

created so that files are never overwritten by later changes
- **project support:** all files associated with a project may be checked out at the same time
- **storage management:** code management system includes efficient storage mechanisms not to keep multiple copies of files that have only small differences

**Two types of source code management:**

- **centralized:** users check in and out files, the system issues warning when checking out files already in use and usually at check in new version of file is created, so system ensure that file copies do not conflicts
- **decentralized:** the most common is git
    - advantages:
        - **resilience:** everyone working on a project has own copy of repository
        - **speed:** committing changes to the repository is fast, local operation
        - **flexibility:** local experimentation is much simpler, developers can safely try different approaches without exposing experiments to others

The principle for which **everything that can be automated should be automated** implies taking advantages of :

- **continuous integration:** producing an executable version at every commits on the master branch
- **continuous delivery:** testing the software in a simulated environment
- **continuous deployment:** releasing new version to users every time a change is made to the master branch
- **infrastructure as a code:** designing a machine-readable model of network, server and routers on which the product executes are used by configuration management tools to build software's execution platform

## Continuous Integration

**Continuous Integration simply means that an integrated version of the system is created and tested every time is pushed to system's shared repository.**
On completion of the push operation the repository sends a message to an integration server to build a new version of the product

## Continuous Delivery

**Continuous Delivery ensures that changed system is ready for delivery to customers: feature tests in production environment to make sure that the environment do not cause system failures and load test are executed to check how software behaves as number of users increases**

**Benefits of CD are:**

- **reduced costs:** fully automated deployment pipeline
- **faster problem solving:** if a problem occurs, it will probably affect only a small part of the system and the source of that problem will be obvious
- **faster customer feedback:** you can deploy new features when they are ready for customer use, hence users feedback can be used to identify improvements
- **A/B testing:** if you have many customer and several servers you can deploy new software version only on some servers, use load balancing techniques to divert some customers to the new version and assess how new features are received

## Infrastructure as Code (IaC)

Manually maintaining a computing infrastructure with tens or hundreds of servers is expensive and error prone.
**IaC allows to automate the process of updating software on servers, by using a machine readable model of the infrastructure. Configuration Management (CM) tools can automatically install software and services on servers according to the infrastructure definition.**
**When changes have to be made, infrastructures model is updated and CM tool makes the changes to all servers.**

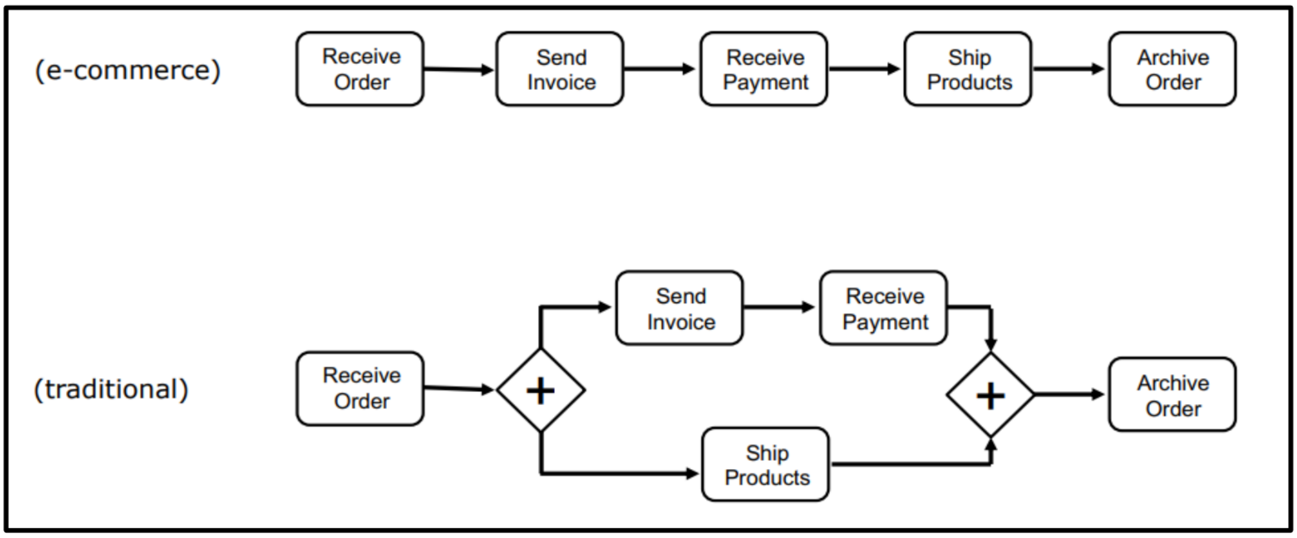
**Advantages:**

- **visibility:** infrastructure defined as stand-alone model that can be read/understood/reviewed by the whole devops team
- **reproducibility:** installation tasks will always be performed in the same sequence, same environment will be always be created
- **reliability:** automation avoids simple mistakes
- **recovery:** infrastructure model can be versioned and stored in a code management system, if infrastructure changes cause problems you can easily revert to older version and reinstall the environment that you know work

# 9. Business Process Modelling

**Business process management is the systematic method of examining your organization's existing business processes and implementing improvements to make your workflows more effective and more efficient.**
**A business process model (BPM) consists of a set of activity models and execution constraints among them them while business process instance (BPI) represents a concrete case in the operational business of a company, consisting of activity instances.**

In figure are pictures of two example of **business process model**



## Business Process Model and Notation - BPMN

**BPMN is constrained to support only the concepts of modelling applicable business processes.**
BPMN model are expressed by simple diagrams constructed from a limited set of graphical elements. For both business users and developers they simplify the understanding of business activities flow and process.

**BPMN four basic element categories:**

- **flow object:** events, activities, gateways
- **connecting objects:** sequence flow, message flow, association
- **swim lanes:** pool, lane
  - pools define a group of participants or external entity
  - lanes defines participant role within the process

- **artifacts:** data object, group, annotation

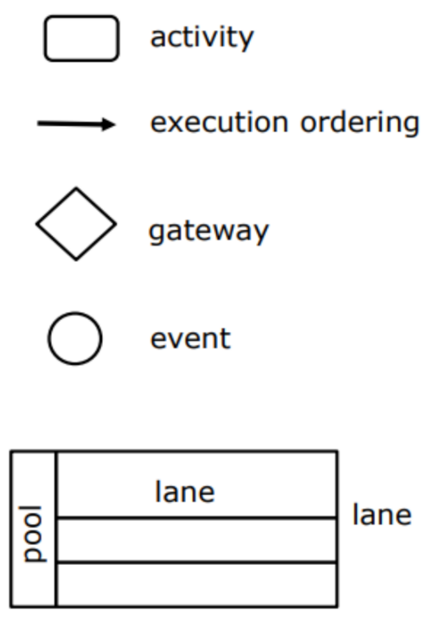
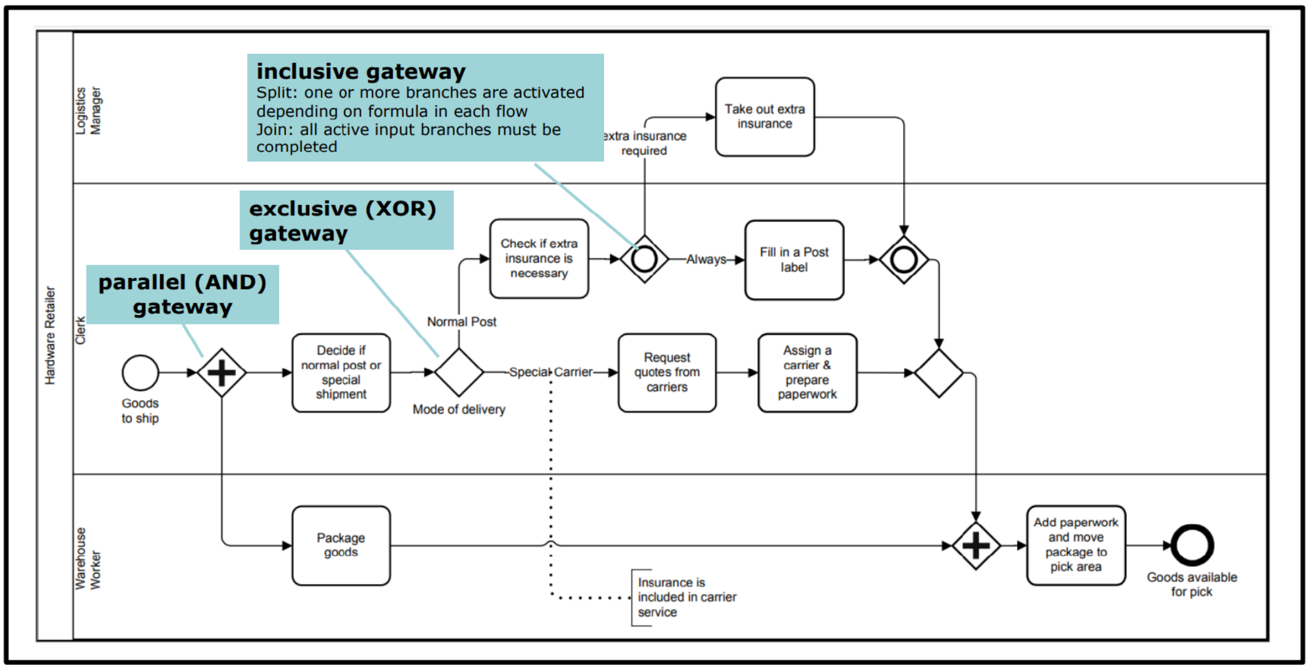


Figure 9.2: BPMN notation

**Complex BPMN Example:**



inclusive gateway
Split: one or more branches are activated depending on formula in each flow
Join: all active input branches must be completed
exclusive (XOR) gateway
parallel (AND) gateway
Hardware Retailer
Logistics Manager
Clerk
Warehouse Worker
Take out extra insurance
extra insurance required
Check if extra insurance is necessary
Always
Fill in a Post label
Normal Post
Goods to ship
Decide if normal post or special shipment
Mode of delivery
Special Carrier
Request quotes from carriers
Assign a carrier & prepare paperwork
Package goods
Insurance is included in carrier service
Add paperwork and move package to pick area
Goods available for pick

# Camunda

**Camunda is a framework supporting BPMN for workflow and process automation.** It provides a RESTful API which allows you to use your language of choice.

# Workflow Nets

Workflow nets are an extension of petri nets, one of the best known techniques for specifying business processes in a formal and abstract way.
Petri nets consists of places, transactions and direct arcs connecting places and transitions.
Transitions model activities, places and arcs model execution constraints.

System dynamics represented by tokens, whose distribution over the places determines the state of the modelled system.
A transition *can fire* if there is a token in each of its input places.
If a transition *fires* one token is remove from each input place and one token is added to each output place.
The idea behind the workflow nets are to enhance petri nets with concepts and notations that ease the representation of business processes.

Like petri nets, workflow nets focus on the control flow behaviour of a process:

- transition represent activities
- places represent conditions
- tokens represent process instances

A petri net is a workflow net if and only if

- there is a unique source place with no incoming edge
- there is a unique sink place with no outgoing edge
- all places and transitions are located on some path from the initial place to the final place

# 10. Testing

**Software Testing is a process in which you execute your program using data that simulates user inputs.**
Tests pass if the behaviour is what you expect, tests fail if the behaviour differs from that expected.

**Many types of testing:**

- **functional testing:** test the functionality of the overall system. the goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose
- **user testing:** test that the software product is useful to and usable by end-users. you need to show that the features of the system help users do what they want to do with the software. you should also show that users understand how to access the software's features and can use these features effectively
- **performance and load testing:** test that the software works quickly and can handle expected load placed on the system by its users. you need to show that the response and processing time of your system is acceptable to end-users. you also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases
- **security testing:** test that the software maintains its integrity and can protect user information from theft and damage

# Functional Testing

**Functional testing involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once.**

Functional testing is a staged activity in which you initially test individual units of code.
**The functional testing processes are the following:**

- **unit testing:** the aim of unit testing is to test program units in isolation. tests should be designed to execute all of the code in a unit at least once. individual code units are tested by the programmer as they are developed
- **feature testing:** code units are integrated to create features. feature test should test all aspects of a feature. all of the programmers who contribute code units to a feature should be involved in its testing
- **system testing:** code units are integrated to create a working version of the system. the aim of system testing is to check that there are no unexpected interactions between the features of the system. system testing may also involve checking the responsiveness, reliability and security of the system
- **release testing:** the system is packaged for release to customer and the release is tested to check that it operates as expected.

## Unit Testing

**A code unit is anything that has a clearly defined responsability. It is usually a function or class method but could be a module that includes a small number of other functions.**
Unit testing is based on the following principle: *if a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for as larger set whose members share these characteristics.*

To test a program efficiently you should identify sets of inputs (**equivalence partitions**) that will be treated in the same way in your code. The equivalence partitions that you identify should not just include those containing inputs that produce the correct values.
You should also identify **incorrectness partitions**, where the inputs are deliberately incorrect.

**Guidelines for Unit Testing**

- **test edge cases:** upper and lower bounds
- **force errors**
- **fill buffers**
- **repeat the same test**
- **overflow and underflow**
- **dont foget null and zero**

- **keep count:** when dealing with lists and list transformations, keep count of the number of elements in each list and check that these are consistent after each transformation
- **test with one-long sequence**

## Feature Testing

**Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.**
Normally, a feature that does several things is implemented by multiple, interacting, program units.

**There are two types of feature test:**

- **interaction tests:** testing interactions between units (developed by different developers) can also reveal bugs in program units that were not expose by unit testing
- **usefulness tests:** testing that feature implements what users are lively to want. product manager should be involved in designing usefulness tests

## System Testing

**System testing involves testing the system as a whole, rather than the individual system features.**

**System testing should focus on four things:**

- testing and discover if there are unexpected and unwanted interactions between the features in a system
- testing to discover if the system features work together effectively to support what users really want to do with the system
- testing the system to make sure it operates in the expected way in the different environments where it will be used
- testing the responsiveness throughput security and other quality attributes of the system

**The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.**
Using the scenario you identify a set of end-to-end pathways that users might follow when using the system.

## Release Testing

**Release testing is a type of system testing where a system that's intended for release to customers is tested.**

The differences between release testing and system testing:

- release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user date, which is sometimes more complex and less reliable than test data
- the aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore some tests that fail may be ignored if these have minimal consequences for most users

## Security Testing

**Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.**
Finding vulnerabilities it is harder than finding bugs.

**A risk based approach to security testing involves identifying common risks and developing tests to demonstrate that the system protects itself form these risks.**

Here **some examples of security risks:**

- unauthorized attacker gains access to a system using authorized credentials
- authorized individuals access resources that are forbidden
- attacker gains access to database using sql poisoning attack
- improper management of http sessions
- ...

## Test Automation

**Automated testing is based on the idea that tests should be executable: an executable test includes the input data to the unit that is being tested, the expected result and check that the unit returns the expected result.**

A good practice is to **structure automated test in 3 parts:**

- **arrange:** set up the system to run the test
- **action:** call the unit that is being tested with the test parameters
- **assert:** assert what should hold if the test are executed successfully

## TDD - Test Driven Development

**Test-Driven Development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.**

Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.

Here are **some pros of TDD:**

- no untested sections
- tests helps understanding the code
- simplified incremental debugging
- arguably simpler code

Here are **some cons of TDD:**

- difficult to apply TDD to system testing
- TDD discourages radical program changes
- TDD leads you to focus on the tests rather than on the problem
- TDD leads you to think about implementation details rather than on overall program structure

## Code Reviews

**Testing have some limitations:**

- you test code against your understanding of what the code should do. If you have misunderstood the purpose of the code, then this will be reflected in both the code and the tests
- testing may not provide coverage of all the code you have written
  - TDD shifts the problem to code incompleteness
- testing do not tell you anything about other attribute of a program (readability, structure, ...)

**Code Reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer.**
Code reviews complement testing: they are effective in finding bugs that arise through misunderstanding and bugs that may only arise when unusual sequences of code are executed.

**Generally one review session should focus on 200-400 lines of code** and should include a checklist with the main checkpoints:

- are meaningful variable and function names used?
- have all data errors been considered and tests written for them?
- are all exception explicitly handled?
- are default function parameters used?
- ...