1. What is Product-Based Software Engineering

Product-Based Software Engineering is another way to approach software development. In this paradigm the starting point for development is a business opportunity, rather than a request from a customer.

Product-Based SE follows these steps:

1. developers/company identify an opportunity
2. the opportunity inspires a set of features to develop
3. the features are implemented in the software product
4. the software product satisfy the identified opportunity

Differently from Project-Based SE in this paradigm the team decides the requirements, has to decide the development timescale and what features include in the product.
The main actor in Product-Based SE is the development team/company.

2. What is the incremental development and delivery advocated by Agile?

In Agile the development is incremental, which means that features are implemented during a short period of time, iteratively.
This time goes from 2 to 6 weeks.
The first measure of success in Agile is the delivery of valuable software, so the objective for every increment is to develop and deliver to the customer an improved version of the product.

3. Which are the key Scrum practices?

The key practices of Scrum are:

- scrums: daily meeting to keep the whole team informed about the progress
- sprints:
  - planning: this phase lasts one day and the objective is to produce a sprint backlog
  - execution: the actual increment take place, in this phase the items in the sprint backlog are implemented
  - review: at the end of every sprint the team reflects about it to see if there are aspects where the process could be improved

4. What are Personas, Scenarios, User Stories and Features?

Personas:

Who are the target for our software product? We need to understand potential users by identifying their background, skills and experience.

Personas are a natural language description of potential users of our product.
Personas allow developers to step into the users shoes.

The aspects of personas descriptions are:

- personalization: give personas name and a bit of context, you should think personas as an individual, not as a role
- 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 relevance
- relevance: if possible it should be stated why this persona might be interested in this product

Scenarios:

A scenario is a narrative description of a situation in which a user is using our product's features to do something that he wants to do.
Scenarios facilitate communication and stimulate creativity, and generally 3-4 scenarios for persona is advised.
They are written from the user's perspective and each team member should individually create scenarios and discuss them with the team.

User Stories:

User Stories are finer-grain narratives that set out in a more detailed and structured way about a single thing that a user wants to do with the product.
User Stories should focus on clearly defined system feature or aspect of a feature that can be implemented in a single sprint.

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

The general template for user story is:

- as a -role-
- I want to do -something-
- so that -reason/values-

5. What is the effect of a git add/branch/clone/checkout/push/commit/pull command?

- git add: updates the index using the current content found in the working tree, to prepare the content staged for the next commit

- git branch: list/create/delete branches
  - git branch: list branches (current branch with an star)
  - git branch -a branch-name: create a new branch
  - git branch -d branch-name: delete a branch
- git clone: clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository, and creates and checks out an initial branch that is forked from the cloned repository's currently active branch.
- git checkout: switch branches or restore working tree files
- git push: updates remote refs using local refs, while sending objects necessary to complete the given refs
- git commit: create a new commit containing the current contents of the index and the given log message describing the changes
- git pull: incorporates changes from a remote repository into the current branch. If the current branch is behind the remote, then by default it will fast-forward the current branch to match the remote

6. How does GitHub flow work?

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.

Workflows are defined in the `.github/workflows` directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks.
For example, you can have one workflow to build and test pull requests, another workflow to deploy your application every time a release is created, and still another workflow that adds a label every time someone opens a new issue.

A workflow must contain the following basic components:

1. One or more events that will trigger the workflow.
2. One or more jobs, each of which will execute on a runner machine and run a series of one or more steps.
3. Each step can either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

7. What is the role of non-functional quality attributes and decomposition in a software architecture?

Non-functional attributes are:

- responsiveness
- performance
- reliability

- security
- resilience
- ...

The architecture of our system greatly affect and is affected by the non-functional attributes.
It is clear that some attribute affect each other and focusing on one can compromise others, a simple example is security-responsiveness: maximal security requires checks and techniques that affects responsiveness.

Decomposing a system is the practice of dividing a software architecture into components, and it is done in according to technology planned to use and in respect of the main non-functional quality attributes that we care the most about.

8. What is a Distribution Architecture?

Distribution Architecture of a software system defines the servers in the system and the allocation of system's components in these servers.

9. Which are the technology choices that affect a software architecture?

The technology choices that affect a software architecture are about:

- database
- delivery platform
- server used
- open source / third party products used
- development tools

10. Which are the main features of Enterprise Integration Patterns?

Enterprise applications are complex distributed multi-service applications whose services must play together suitable integrated.

The architectural question is how to integrate multiple different services to realize enterprise applications that are coherent (by understanding each other), extensible, maintainable and reasonably simple to understand.
This is the key idea behind the enterprise application integration patterns that allows to manage the complexity behind every system, tolerate the changes and be a unified model by introducing pattern methods.

A pattern is an high-level abstraction of accepted, reusable solutions to recurring problems that saves us from reinventing the wheel and making the same mistakes as others.

11. Which are the differences between multi-tenant and multi-instance SaaS systems?

- **multi-instance system:** 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 than multi-tenant system and avoid security concerns such as data leakage from one organization to another
- **multi-tenant system:** software architecture in which a single instance of software runs on a server and serves multiple tenants. Systems designed in such manner are "shared" (rather than "dedicated" or "isolated"). A tenant is a group of users who share a common access with specific privileges to the software instance. With a multi-tenant architecture, a software application is designed to provide every tenant a dedicated share of the instance - including its data, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants.

12. What is the effect of docker build/run/commit/tag?

- docker build: builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context
- docker run: runs a command in a new container, pulling the image if needed and starting the container.
- docker commit: commit a container’s file changes or settings into a new image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server.
- docker tag: after the image name, the optional TAG is a custom, human-readable manifest identifier that is typically a specific version or variant of an image.

13. What is Docker Compose?

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

14. How does K8s control plane work?

K8s' control plane make global decisions about the cluster (e.g.: scheduling) as well as detecting and responding to cluster events (e.g.: starting a new pod)

In the cluster there are two types of machines:

- master node: (often a single) machines that contains the components of the control plane
- worker node: machines that run the app's workload

In the master node we find the components of the control plane:

- api-server: fronted of the control plane, it validates user requests and acts as a unified interface for questions about the cluster state
- etcd: key-value database in which is stored the cluster database
- scheduler: determines where pods should run
  - asks the api-server which Pod has not an assigned machine (node)
  - determines which machine to use
  - give the response to the api-server
- control-manager: monitor cluster state (through api-server). If the state differs from the desired state it will take action.

15. What is Minikube?

Minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.
Minikube implements a local Kubernetes cluster on macOS, Linux, and Windows.

Minikube's primary goals are to be the best tool for local Kubernetes application development and to support all Kubernetes features that fit.

16. Which are the main pros, cons and characteristics of Microservices?

Main Characteristics

- componentization via services: the application get decomposed in services, which are independently deployable
- microservices allow for cross-functional team-based organization: every team has a full range of competences
- business-oriented
- products non projects: allows the "you build it you run it" philosophy
- self-contained: microservices do not have external dependency

Pros

- lightweight
- independently deployable
- effective scaling
- fault resiliency
- implementation independent

Cons

- complexity

- services will need to share data, and it is not always feasible to wrap these data in service and provide an API, due to exceeding time/complexity. The solution resides in data duplication, which is useful for microservice isolation but problematic in terms of consistency.
- a poor team results in a poor system
- hard time monitoring
- security: a more ample attack surface than monolithic systems

17. >> What does the CAP theorem tell us?

The CAP theorem is a key result in distributed system.
It states that, in presence of a network partition, we can not have both availability and consistency, where:

- consistency: any read operation that starts after a write operation must return the update value
- availability: every request received from a non-failing node must result in a response
- network partition: network can loose any messages sent from one group to another

No distributed system is safe from network failures, thus network partitioning generally has to be tolerated.
In the presence of a partition, one is then left with two options: consistency or availability.
When choosing consistency over availability, the system will return an error or a timeout if particular information cannot be guaranteed to be up to date due to network partitioning.
When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

18. >> Which refactoring can be applied to resolve architectural smell x?

The design principles to follow to avoid architectural smells are:

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

Here the list of smells and how to resolve them:

- smell: **no api gateaway**

    - description: the external clients of an application directly interact with internal service. If one of such service is scaled out the horizontal scalability may be violated because the external clients may keep invoking the same instance
    - refactoring: add an api gateway message router
- smell: **shared persistency**
    - description: the shared persistency smell occurs whenever multiple services access or manage the same DB, possibly violating the decentralization principle
    - refactoring: add a data manager
- smell: **wobbly service interaction**
    - description: an interaction between two services is wobbly when a failure of a service can trigger the failure in the other service
    - refactoring: use timeout
- smell: **endpoint-based interaction**
    - description: one or more microservice invoke a specific instance of another microservice (e.g.: because it is hardcoded in their source code)
    - refactoring: add a service discovery

19. How can K8s or Swarm resolve architectural smells?

K8s api-service prevents the endpoint-based interaction since all the communication goes through an unified interface

20. How can we feature authentication and authorization in a software product?

Authentication

Authentication is the process of ensuring that a user of your system is who they claim to be.
There are many approach to authenticate an user:

- knowledge based: authentication relies on something that the user know
- possession based: authentication relies on something that the user has
- attribute based: authentication relies on something that the user is

Authorization

Authorization is a complementary process of authentication in which the identity is used to access the software system resources and functionalities.

This policy is a set of rules that define how and which aspects of the system (functionalities and data) can be used by authenticated users.

Access Control Lists are how the access control policy is implemented:

- classify users into groups (helps reduce the ACL size)
- different group can have different rights
- hierarchies of groups to assign rights to subgroups/individuals

Providing Authentication and Authorization

Authentication can be provided by an hand-made authentication system or through federated identity
Federated Identity is an approach to authentication where you use an external authentication service, such as "login with google".
The advantage is that users have to remember a single set of credentials that are stored by a trusted identity service.
The main drawback for the system's owners is that they have to share the credentials of their users with the third party authentication provided.

Authorization can be provided by the use of the OAuth2.0 protocol.
OAuth 2.0 is an authorization protocol and it is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user data.
OAuth 2.0 uses Access Tokens. An Access Token is a piece of data that represents the authorization to access resources on behalf of the end-user. OAuth 2.0 doesn't define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used. This enables token issuers to include data in the token itself. Also, for security reasons, Access Tokens may have an expiration date.

21. Which are the main challenges in securing microservices?

In a microservice architecture each microservice has to carry out independent security screening that may need to connect to a remote security token service.

There are many challenges in securing microservices:

- the attack surface is broader than the monolithic architecture, hence the risk is higher
- 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 responsibility is shared among different teams

22. >> Which are the most frequent API security vulnerabilities and how can we prevent them?

The most frequent API vulnerabilities identified by the OWASP

- **broken object-level authorization** x

    - description: attacker substitutes ID of their resource in a API call with an ID of a resource belonging to another user. Lack of proper authorization checks allow access
    - solutions:
        - implement authorization checks with user policies and hierarchy
        - don't rely on IDs sent from the client, use IDs stored in the session object instead
        - use random non-guessable IDs
        - ...
- **broken authentication** x
    - description:
        - poorly implemented API authentication allowing attackers to assume other user's identities
        - solutions:
            - use short-lived access tokens
            - use standard authentication, token generation, multi-factor authentication
            - ...
- **excessive data exposure** x
    - description:
        - API exposing a lot more data than the client legitimately needs, relying on the client to do the filtering. The attacker goes directly to the API and has it all
    - solutions:
        - never rely on the client to filter data
        - define schemas of all the API response
- **lack of resources & rate limiting** x
    - description:
        - API is not protected against an excessive amount of calls or payload sizes. Attackers use that for DoS and brute force attacks.
    - solutions:
        - rate limiting
        - payload size limits
        - rate limits specific to API methods, clients, and addresses
- **broken function level authorization** x
    - description:
        - API relies on client to use user level or admin level APIs. The attacker figures out the hidden admin API methods and invokes them directly.
    - solutions:
        - don't rely on app to enforce admin access
        - deny all access by default
        - grant access based on specific roles
        - properly design and test authorization
- **security misconfiguration** x

    - description:
        - poor configuration of the API servers allows attackers to exploit them.
    - solutions:
        - repeatable hardening and patching processes
        - automated process to locate configuration flaws
        - disable unnecessary flaws
- **mass assignment** x
    - description:
        - API working directly with the data structures
        - Received payload is blindly transformed into an object and stored
        - Attackers can guess the fields by looking at the GET request data
    - solutions:
        - don't automatically bind incoming data and internal objects
        - explicitly define all the parameters and payloads you are expecting
        - for object schemas use the readOnly set to true for all properties that can be retrieved via API but should never be modified
- **injection** x
    - description:
        - attacker constructs API calls that include SQL, NoSQL, and other commands that the API or the backend blindly executes
    - solutions:
        - never trust your API consumers
        - strictly define all input data: schemas, string patterns, ... and enforce them at runtime
        - validate, filter and sanitize all incoming data
- **improper asset management** x
    - description:
        - attacker finds non production versions of the API (staging, testing or early versions) that are not as well protected and use those to launch an attack
    - solutions:
        - inventory all API hosts
        - limit access to anything that should not be public
        - implement additional controls such as API firewalls
- **insufficient logging & monitoring**
    - description:
        - lack of proper logging, monitoring and alerting let attacks go unnoticed
    - solution:
        - log failed attempts, denied access, input validation failures, ...
        - ensure that logs are formatted to be consumable by other tools
        - protect the logs as highly sensitive data
        - include enough info to identify attackers
        - avoid having sensitive data in logs

23. What is DevOps Automation?

DevOps integrates development, deployment and support in a single team.
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 principle for which everything that can be automated should be automated implies taking advantages of:

- continuous integration: producing an executable version at every commit 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 the network, server and routers on which the product executes so that the configuration management tool can build/update/manage the software's execution platform

24. What is Jenkins?

Jenkins is an open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery.
It is a server-based system that runs in servlet containers such as Apache Tomcat. It supports version control tools.

25. How does Jenkins exploit Git?

The git plugin provides fundamental git operations for Jenkins projects. It can poll, fetch, checkout, branch, list, merge, tag, and push repositories.

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

**Gateways are used to control how sequence flows interact as they converge and diverge within a process.
If the flow does not need to be controlled then a gateway is not needed.

The term "gateway" implies that there is a gating mechanism that either allows or disallows passage through the gateway**: as tokens arrive at gateways they can be

merged together on input and/or split apart on output as the gateway mechanisms are invoked.

A gateway is graphically displayed as a diamond.

- **exclusive gateway:** an exclusion gateway (decision) is used to create alternative paths within a process flow. For a given instance of the process only one of the paths can be taken
- **inclusive gateway:** a diverging inclusive gateway (inclusive decision) can be used to create alternative but also parallel paths within a process flow. Unlike the exclusive gateway all condition expression are evaluated
- **parallel gateway:** a parallel gateway is used to synchronize (combine) parallel flows and to create parallel flows. A parallel gateway creates parallel paths without checking any conditions
- **event-based gateway:** an event-based gateway must have at least two outgoing sequence flows. Each sequence flow must to be connected to an intermediate catch event of type timer or message. When an event-based gateway is entered, the process instance waits at the gateway until one of the events is triggered. When the first event is triggered, the outgoing sequence flow of this event is taken. No other events of the gateway can be triggered afterward.

Memory trick:

- **parallel = AND, kinda**
- **inclusive = OR**
- **exclusive = XOR**

27. >> What is a Workflow Net? What is a sound/live/bounded net?

Workflow Net

Workflow nets are petri-net based workflows that are suitable for expressing workflows. A petri net is a mathematical modelling language for the description of distributed system.

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 paths from the initial place and the final place

Sound/Live/Bounded Workflow Net

A workflow net is sound iff

- is bound: every net execution starting from the initial marking (one token in the initial place, no tokens elsewhere) eventually leads to the final marking (one token in the final place, no tokens elsewhere)
- is live: every transition occurs in at least one net execution
- A Petri net is **live** if for any possible reachable state, any transition can possibly fire in the future.
- A Petri net is **bounded** if all the places can't have more than a fixed number of tokens, considering them in all the possible future states.

28. What is Camunda?

Camunda is an Open-source workflow and decision automation platform, used for microservices orchestration.
The workflows are defined in BPMN which can be graphically modeled using Camunda Modeler.
Camunda runs as a separated microservice and provides RESTful API to configure and make use of it.

29. >> Which are the two "usage patterns" of Camunda?

1. **Pattern A: endpoint-based integration**
    1. each part of the BPM can call a microservice by configuring a connector
    2. easy to configure (just specify url, type of request, headers)
    3. microservice can't scale (camunda will always connect to a single instance of the microservice)
2. **Pattern B: queue-based integration**
    1. each part of the BPM put requests inside a topic queue. Workers pull requests from queue and interact with instances of the microservices (using load-balancing)
    2. it is harder to configure (workers must subscribe to a specific topic)
    3. scales very well (process instances of Camunda can scale, as well as workers and microservices)

30. What is 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 the 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 units of code are tested by they programmer as they are developed

- **feature testing:** code units are integrated to create features. features should test all aspects of a feature. all 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 are no unexpected behavior
- **release testing:** the system is packaged for release to customers and the release is tested to check that it operates as expected

31. What is Test Automation?

Automated Testing is based on the idea that tests should be automatically executable (potentially at every push in the remote repository or any other given event).
An executable test includes the input data, the expected result and checks (assertions) that verify that the result is the expected one.

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

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

33. What is Locust?

Locust is a simple Python-based load test tool. It can run with and without web UI, also it supports standalone and master/slave modes. User can write tests with a small set of APIs provided by Locust, and then run the tests in Locust by specifying number of tests to run, the speed of test etc.

34. How can you use it to identify bottlenecks?

Testing specific microservices

35. >> Which are the problems of and solutions for application management in the Cloud-Edge Continuum?

Problem 1: How to Suitably Place Composite Applications on the Cloud-Edge Continuum

There are many approach, we are more interested in the declarative one:

- declare what an eligible placement is
  - service S can be placed on the node N if
    - the hardware reqs of S are met by N and
    - ...
  - service S1 can ...
- let the inference engine look for it

Problem 2: How to Suitably Manage Application Deployments in the Cloud-Edge Continuum

Continuous reasoning: exploit compositionality to differentially analyse a large-scale system:

- focus on the latest changes introduced by the system
- reuse previously computed results as much as possible

36. What is an explainable failure root cause analysis?

Cascading failures may occur in microservice-based applications. Determining the root cause of a cascading failure is crucial and very hard (beware of the correlation is not causation bias).

The availability of explanation of the cascading failure would permit to avoid further failure by intervening on:

- the failing service
- the services that fails consequentially

An approach to root cause analysis in microservices architecture is the declarative approach:

- define casual relations between events:
  - if service S logged a failure/timeout event E at the end of the interaction with I and I logged an internal error during the interaction
  - then add E to the explanation and analyze the cause of internal error of I
- let the inference engine work

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

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

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

38. Which are the pros and cons of REST?

REST Pros

REST is very simple and human intuitive, more scalable and efficient than others standards/protocols like SOAP. The learning curve is very low since there are lots of experimented tools that allow to turn existing applications into RESTful services by

providing and exposing HTTP APIs.
The exploit of a stateless protocol like HTTP allows for a great scalability degree.
This also allows for an easier testing/development phase (by using a web browser or tools like postman)

REST Cons

On the other hand, complex interfaces are hard to manipulate via URLs, both because of their mixed length and because the resource tree to be designed might become very complex.
Also, there is no best practice manual to design good REST APIs, and no semantic rules are provided in order to fix the functionalities of HTTP methods, so potentially one service could make a resource deletable via a GET method.
QoS management and monitoring must be implemented "by hand" and the design of resource specification and URI addressing scheme might become challenging.

Conclusion

REST performs well for human application (humans read manuals), while for machine readable and understandable applications and for enterprise applications, other techniques might be a better choice.

39. Why Microservices?

Microservices:

- small-scale services that can be combined to create applications
- independent (service interface not affected by changes of other services)
- possible to modify and re-deploy service without changing/stopping other services

Microservices Characteristics:

- self-contained: microservices do not have external dependencies: manage their own data and implement their own UI
- lightweight: communication through lightweight protocols
- implementation independents: microservices in the same system may use different technologies
- independently deployable: each microservice runs in its own process
- business oriented: microservices should implement business capabilities needs

Three main concepts:

- low coupling: coupling measures the number of inter-component relationships, hence low coupling results in independent services: independently updatable, deployable, ...
- high cohesion: cohesion measures the number of intra-component relationships, hence high cohesion means that there is less communication overhead between inter-services

Why Microservices?

- shorten lead time for new features and updates, accelerate rebuild and redeployment
- scale, effectively

40. What is a container/image/volume in Docker?

- 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
- container: containers are executing images. An image is loaded into a container and the application defined by the image starts execution
- volume: volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

41. Which are the differences between a virtual machine and a container?

The key differentiator between containers and virtual machines is that virtual machines virtualize an entire machine down to the hardware layers and containers only virtualize software layers above the operating system level.

42. What is development/release/user testing?

Software testing is a process in which you execute your program using data that simulates user inputs.
Tests pass if the behavior of the system is the expected one, otherwise the tests fails.

- **development testing:** the system is tested during the development to discover bugs and defects
- **release testing:** testing of a complete version of the system before it gets released to users, the aim is to check that the system meets the given requirements
- **user testing:** a restricted number of real users test the system in their own environment

43. What is TDD?

Test Driven Development is an approach to program development that is based around the 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, it is more difficult to apply it to system testing

TDD pros:

- no untested sections

- testing helps understand code
- simplified incremental debugging
- arguably simpler code

TDD cons:

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

44. What is partition testing?

There are two main strategy to compare when we talk about unit testing effectiveness

- partition testing: identify groups of inputs that should be processed the same and choose tests from within each of these groups
- guideline based testing: choose test cases by using testing guidelines (that reflect previous experience of the kinds of errors that programmers often makes)

In partitioning testing we identify equivalence partitions - classes of inputs (and output) data that have the "same characteristics" and for which the program normally behave in a comparable way, then we create test cases from each partition in order to test boundaries input to our functionalities

45. >> Which are the main "smells" un microservices security?

10 refactoring associated to 10 smells that potentially affect 3 security properties.

Security Properties: CIA

- confidentiality: access to data is granted only to authorized individuals
- integrity: prevents unauthorized modification of program and/or data
- authenticity: identity can be proved to be the one claimed
- **smell:** insufficient access control
  - **property affected:** confidentiality
  - **refactoring:** use OAuth 2.0 protocol to properly authenticate users
- **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 and never hardcode secrets (use KMS, ..)
- **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
- **smell:** multiple user authentication
    - **properties affected:** authenticity
    - **refactoring:** add API-gateway, use OpenID connect and use single sing-on
- **smell:** centralized authorization
    - **properties affected:** authenticity
    - **refactoring:** use decentralized authorization
        - solution is represented by enacting a decentralized authorization approach: transmit an access token (e.g. JWT) together with each request to a microservice and access to microservice is granted to caller only if a known and correct token is passed.

46. What is Kubernetes?

Kubernetes is a container orchestration platform.
Kubernetes, is an open-source system for automating deployment, scaling, and management of containerized applications.

47. Which properties should be enforced to guarantee software security?

The CIA property:

- confidentiality
- integrity
- authenticity

48. What is REST?

REST (REpresentational State Transfer) was proposed as an architectural style which featured the network of web pages as a "distributed" finite state machine, where actions result in a transition to the next state.

REST is, in practice, a redesign of the HTTP protocol in order to feature:

- resource identification through URIs: the server exposes a set of resources identified by predefined URIs
- uniform interfaces: the possible operations are just the HTTP ones (PUT, GET, POST and DELETE)
- self descriptive messages: the approach is resource centric, resources are decoupled from their representation so that their content can be processed in a variety of formats (JSON, XML, ...)
- stateful interactions through URIs: HTTP is a stateless protocol, but a stateful interaction can be achieved via responding to requests (on the server side), with a new state

49. What is Git/GitHub?

Git

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

GitHub

The GitHub platform is a web-based hosting service for version control using git. GitHub hosts a shared repository where one or more users can collaborate.
Thanks to plugins in github is possible to deliver software with CI/CD techniques.

50. >> What is the Model-View-Controller Pattern?

The model-view-controller patter consists on three components:

- model: the entity that manage the data
- view: the component that display the data
- controller: the entity that manipulates the model

The view, which is the component that the client uses, catches user events and forwards them to the controller. The controller then maps these events to state changes in the model. The model needs to inform the view of every change, and the view can query the model in order to retrieve state info.
The controller can also select the view to be shown.

51. What are the service/deployment models of Cloud Computing?

(ICT Infrastructures 101)

Service Models

- Software as a Service (SaaS)
- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)

Deployment Models

- private cloud
- public cloud
- hybrid cloud

52. What is a Product Vision?

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

- what is the product
- who are the target customers
- why the customer should buy/use the product

The product vison is determined by:

- domain experience: the developers work in a particular area and understand the software support they need
- customer experience: developers may have extensive discussion with perspective customers
- product experience: users of existing product may see simpler and better ways to do things
- prototyping and playing around: useful to understand better the product and technologies involved

Product Vision general template:

- for "target customers"
- who "need or opportunity"
- the "product name" is a "product category"
- that "key benefit"
- unlike "main competitor"
- our product "statement of primary differentiation"

53. What are Code Reviews

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 very effective in founding bugs that arise through misunderstanding of the objective and bugs that arise only when an unusual sequence of code is executed.

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

- are variables and functions named significantly?
- have data errors been considered and tests written?
- are all exception handled?
- are default functions parameters used?
- ...

54. >> Describe the Microservices Architecture

A microservice 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 the most popular system functions

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

55. >> Why System Testing is hard?

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

System testing should focus on four things:

- testing to 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 environment where it will be used
- testing the responsiveness, throughput, security and other quality attributes in 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: an end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

56. >> Direct vs Indirect Communication

- **direct communication:** A and B send messages to each other
- **indirect communication:** A and B communicate through a message broker

57. >> Describe the Saga Pattern

The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios.
A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

This introduces the necessity of introducing a mechanism for compensating transactions. Two models are available

- **backward model:** undo changes made by previusly executed local transactions
- **forward model:** "retry later" at regular intervals

Two ways of coordinating sagas:

- **choreography:** each local transaction publishes event that triggers next local transactions
- **orchestration:** an orchestrator tells partecipants which local transactions to execute

58. What is the Jenkins Keep Up Flow?

TODO BOH

59. Continuous Integration: Reason to NOT Deploy?

If a system is infrequently integrated, problems can be difficult to isolate, and fixing them slows down system development: continuous integration should be used.

An integrated version of the system is created and tested every time a change is pushed to the system's shared code repository.

On completion of the push operation, repository sends message to integration server to build a new version of the product.

Adopt an “integrate twice” approach to system integration:

- integrate and test on your own computer
- push code to project repository to trigger integration server

Why not deploy? Don't know: Improvise, adapt, overcome.