.NET & Azure Interview

**Checklist**

# 1.    .NET

**☐**   Async vs Parallel

**Cooperative multitasking vs Physical Multitasking**

**☐**   Tasks Run vs Async / Await

**Pausing Current Thread vs Continuation State Machine**

**☐**   Concurrency Management

**Race Conditions**

**Dead Lock**

**.NET provided solutions (lock, Monitor, Interlock, Semaphore, Reset Events, Wait Handles, Mutexes)**

**☐**   Cancellations / Continuations

**When To Use them**

**How to combine them**

**☐**   Iterators

**How to implement it**

**How to do Streaming (AsyncEnumerable)**

**☐**   Events and Delegates

**When to use them**

# 2.   Automated Unit testing

**☐**   Test Driven Development

**☐**   Test Doubles (Stub, Mock, Fake, Dummy, Spy)

[https://www.techtarget.com/searchsoftwarequality/tip/Inside-5-types-of-test-doubles](https://protect-us.mimecast.com/s/a-FWC73z4wfzP962T8HN8m)

How to use Mocks in Testing

**☐**   Test Types

[https://www.perfecto.io/resources/types-of-testing](https://protect-us.mimecast.com/s/rougC82A4xuXJxMWC1TtKL)

**☐**   Unit Test Alternatives

**Behavior Testing vs Data Testing**

**When to use each one**

**☐**   Code Quality Standards and Tools

**SonarQube**

# 3.   Azure

**☐**   How to select a messaging solution

**☐**   Messages vs Events

**Message = Contract, Self-Contained, Expectations on the Consumer Side**

**Event = Signal, Does not Contain Data, No Contract or Expectations**

**☐**   How to select a database solution

**☐**   SQL vs NO SQL

**ACID (Atomicity, Isolation, Consistency, Durability)**

**CAP Theorem (Consistency, Availability, Page Fault Tolerance)**

**SQL = Consistent, Transactional, Requires Data Normalization**

**No SQL = Eventually Consistent, Usually Single Document Transactional, should use soft-lock solutions, Encourages Data Denormalization**

# 4.   DEVOPS and source conrol

# 

**☐**   Distributed vs Centralized

**Git vs TFVC**

**☐**   Distributed Source Control Alternatives

**Trunk based vs Branch based**

[**https://lucamezzalira.com/2014/03/10/git-flow-vs-github-flow/**](https://protect-us.mimecast.com/s/Akc8C9rB4yIzZPKAT3cWz0)

**Strategies, Git flow, GitHub flow**

**Git flow** works with different branches to manage easily each phase of the software development, it’s suggested to be used when your software has the**concept of “release”** because, as you can see in the scheme above, it’s not the best decision when you work in Continuous Delivery or Continuous Deployment environment where this concept is missing.

**GitHub** is working with a continuous deployment environment where there isn’t the concept of “**release**” because every time they finish to prepare a new feature they push live immediately (after the whole [automation chain](https://protect-us.mimecast.com/s/r96CC0RpQmIkElWQHWw6VE) created in the environment

**☐**   CI/CD

-          **Continuous Integration**

-          **Continuous Delivery vs Continuous Deployment**

# 

# 5.   design Patterns and best practices

**☐**   Design Patterns

[https://en.wikipedia.org/wiki/Software\_design\_pattern](https://protect-us.mimecast.com/s/nOmJCgJKzGTqgnWKfElwfB)

[https://opensource.com/article/19/7/understanding-software-design-patterns](https://protect-us.mimecast.com/s/6Vy8CjRMDkIY6kJDUn37sd)

**What they are**

**According to Wikipedia, "a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design."**

**Types (Structural, Behavioral, Data Mapping)**

**Samples**

**☐**   Domain Driven Design

[https://en.wikipedia.org/wiki/Domain-driven\_design](https://protect-us.mimecast.com/s/5j92CkRKElIkLZ0AHk3SVG)

**Domain-driven design** (**DDD**) is a [software design](https://protect-us.mimecast.com/s/aSCOClYLGmC19DKmfgDUKT) approach [[1]](https://protect-us.mimecast.com/s/xVdrCmZ6Jnup0GY8f45rzL) focusing on modelling software to match a [domain](https://protect-us.mimecast.com/s/YjxsCn5XKotXyQLgCPRCmR) according to input from that domain's experts.

**☐**   Continuous Integration, Delivery, Deployment

**Differences between Delivery and Deployment**

**☐**   SOLID Principles

[**https://stackify.com/solid-design-liskov-substitution-principle/**](https://protect-us.mimecast.com/s/sfQWCo2KL0uDoJG5CE0snW)

**Single Responsibility, Open-Close, Liskov Substitution, Interface Segregation, Dependency Inversion**

**☐**   Code Smells

[**https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/**](https://protect-us.mimecast.com/s/X9TLCpYKMqC9lPLqi3jPx9)

**Code smells** are not bugs or errors. Instead, these are absolute **violations** of the **fundamentals of developing** **software** that decrease the quality of code.

**☐**   Microservices

**Architectural style that approaches building applications as a suite of services, each running in its own process, communicating through lightweight mechanisms like simple HTTP or messaging.**

**☐**   Serverless

[**https://www.datadoghq.com/knowledge-center/serverless-architecture/#:~:text=Serverless%20architecture%20is%20an%20approach,storage%20systems%20at%20any%20scale**](https://protect-us.mimecast.com/s/J5kACqx5NrI14GrnfWhtDf)**.**

**Serverless architecture is an approach to software design that allows developers to build and run services without having to manage the underlying infrastructure. Developers can write and deploy code, while a cloud provider provisions servers to run their applications, databases, and storage systems at any scale.**

**Samples**

**☐**   Closures

**First class function with private variables bound to a lexical context.**

**Functions with access to variables from another function’s context.**

**☐**          DevOps

**DevOps is a set of practices that combines software development and IT operations. It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. DevOps is complementary with Agile software development; several DevOps aspects came from the Agile methodology**