**Message queues**

1. What is a message queue? What do message queues store and transfer?

A message queue is a software component used for communication between different processes or threads, (often on different machines/applications). It's used in an asynchronous method of communication that saves the response for future processing, offering a protocol for sending and receiving messages.

Message queues store and transfer **messages** – data structures that contain information, like instructions or process information, to be communicated between different processes. The data can be in any format (like JSON, XML, plain text, binary data etc.), containing any kind of the details (e.g. info about the receiver, sender, message body, priority level).

1. Describe the publisher/subscriber pattern. The difference between Pub/Sub and Observable patterns.

The publisher/subscriber pattern (Pub/Sub) is a messaging pattern where senders of messages, called **publishers**, do not managed to send the messages directly to receivers, known as **subscribers**, so the message is produced (sent) to the end subscriber which is never known by publisher (he even may not exist at all). Subscribers represent the dependent consumers of information, and their only interest is to receive specific messages, without knowledge of which publishers, if any, there are.

The main difference between a Pub/Sub and an Observable pattern lies in **the way they handle data** and **the type of systems they are designed for**.

Pub/Sub is used for distributed systems (publishers can send messages without knowing if there are any subscribers to receive those messages; similarly, subscribers will receive messages without knowing who published them - that provides a level of *decoupling* between the pub & the sub)

Observable pattern is used for handling events or streams of data: the observable object holds the list of observers, so any changes in the state of the observable object make the observer handle that (all the associated observers get notified). Observers subscribe directly to the observable.

One way to think of the key difference is that the Observable pattern is **pull-based** (the observer controls when it receives data from the observable), while the Pub/Sub pattern is **push-based** (the publisher sends data to the subscriber when new data is available).

1. What is a Message Bus? How does it work?

A message bus is a kind of software communication tool used to exchange information between applications or services within a system (or across systems). It operates with a common communication channel that enables different software components to communicate, even if they are written in various programming languages and run on separate machines.

The principal components in a message bus system are **the sender** and **the receiver**. The sender sends the messages to the bus, and the receiver fetches (gets notified) about the messages from the bus.

Based on the information above, the way it works is following:

* A *system* sends a message to the message bus (this *system* is usually known as the producer or the sender)
* The message bus receives the message and places it in a queue for processing, ensuring that it doesn't get lost if the receiver is currently unavailable or too busy.
* Another system, known as the consumer or receiver, connects to the message bus and retrieves the message.
* The receiver then processes the message in whatever way it needs.

1. What is the difference between message queue and web services?

Message queue is a form of **asynchronous** service-to-service communication used in serverless and microservices architectures. It represents a list of messages sent between applications or within processes to execute certain tasks. It follows a first-in, first-out (FIFO) model and helps decouple sending and receiving applications, allowing messages to stack up or be **delayed** without crashing the system.

Web service is a set of standards (protocols) for exchanging information between different systems over the internet. It allows different applications to communicate with each other in a platform-independent and language-independent way. Web services use HTTP protocol for the communication (**synchronous)**, and the data is usually formatted as XML/JSON. The client **waits for the server to respond**, allowing for live, real-time interaction or data exchange

1. Describe the difference between RabbitMQ and Kafka. Provide some use cases for each of them: in which scenarios you’ll use RabbitMQ, Kafka?

RabbitMQ is a message broker with a focus on a variety of messaging patterns, including point-to-point, request/reply, and pub-sub. It has a rich set of features, such as reliable delivery, flexible routing, clustering, and high availability.

Kafka was designed as a message bus optimized for high ingress data streams and replay, with the ability to handle real-time data feeds. It maintains feeds of messages in categories called topics, and ensures that messages are retained for some period.

RabbitMQ use case: the application needs **a variety of messaging patterns** beyond just pub-sub or streaming – in this case RabbitMQ is a good choice. It would also work well if there are **some tasks to be processed asynchronously** in the background. Besides that, RabbitMQ has flexible routing mechanism, so the **complex routing to multiple receivers** may be developed.

Kafka use case: in situations where it’s important to process huge volumes of real-time data (or it’s necessary to implement real-time analytics and logging, where preserving the order of messages is important). Kafka may help in case of replay capabilities are needed (for example, a new consumer needs old data *or* some existing consumer goes down and needs to reprocess data when it comes back)

**Troubleshooting & Logging**

1. What are the differences between performance, load, and stress testing?

All of those are the methods used to evaluate the functionality and endurance of a system under different demands and pressures.

**Performance Testing** is a kind of testing to evaluate how a system performs under certain loads or the speed at which a system accomplishes its function. It focuses on ensuring that the **application's response times meet the expected standards** and that it can handle the expected concurrent user load.

**Load Testing** checks how a system behaves and performs under a specific load, mainly high traffic or data load. For software applications, this type of testing could involve assessing **how the system copes with large volumes of users or transactions**.

**Stress Testing** – the most intense type of testing. It aims **to break the system** **by applying overwhelming demand**, often beyond the system's stated capacity. These tests observe **how the system recovers when the load returns to typical levels**. Stress testing can help identify bugs that only show up under extreme conditions or point out potential vulnerabilities that could be exploited by malicious attacks.

1. When would you prefer vertical scaling over horizontal?

Vertical means increasing power capabilities of existing machines to meet demand (e.g. upgrading CPU, memory, storage). This type of scalability is preferred in cases of **predictable growth of demand**, easy for quick prototyping (there is only one unit that doesn’t need to be coordinated).

Horizontal means adding nodes (machines) to the infrastructure to handle increased demands. This is an analogy of real spread of work between employees (same to nodes), so the load and traffic may be distributed effectively. But here it’s important to remember about correct managing the nodes and their communication.

1. Does ASP.NET Core API support horizontal scaling? Explain your answer.

Yes, it does.

Horizontal scaling presumes scaling by adding more machines into the pool of resources (at the same time Vertical scaling is adding more power (CPU, RAM) to the existing machine).

The application or API should be able to *handle multiple requests coming from different machines or instances*. ASP.NET Core, as a fully-featured, scalable framework, is designed keeping that in mind: the stateless nature of the HTTP protocol and the built-in functionality of ASP.NET Core to handle such requests, the efficiency of deploying and running containerized applications (which is a common strategy in horizontal scaling), the support of layered architecture (different middleware, including DI, logging, configuration, etc.) – these are the main facts that prove ASP.NET Core API is a good choice for the applications that need horizontal scaling.

One of the ways to achieve horizontal scaling in ASP.NET Core API is to use cloud services that support that type of scaling (e.g. Azure Container Apps) or to implement a custom load balancer (e.g. YARP – Yet Another Reverse Proxy – a high-performance reverse proxy library from Microsoft, which will work as traffic director).

**Caching and Multithreading**

1. How ASP.NET API handles multiple requests?

ASP.NET API handles multiple requests concurrently by creating a new instance of the controller for each request. It is inherently multi-threaded, so it is designed to handle multiple requests at the same time.

ASP.NET applications (independently from being hosted in IIS or Kestrel) create separate worker processes to handle incoming HTTP requests. Each of these worker processes operates independently to process requests. If two requests arrive simultaneously, they use might use two different worker processes and those requests would get executed concurrently. In addition, each request is usually handled by its own thread – simultaneous requests can be handled in parallel, improving the overall performance of the application. This process is managed by the .NET Thread pool, which keeps track of the number of requests being processed and adjusts the number of available threads as necessary.

1. What are the benefits and downsides of caching? When should we consider applying caching?

+ allows to improve performance because of storing data closer to where it's needed (like in memory or local storage), without time-consuming operations like going through all the layers of the system to the database

+ without resource-consuming operations (complex queries, etc) the load of the server is reduced

+ increased reliability: sometimes database can have problems like failures, cached data is always available (of course, depending on the expiration time of the cached items)

* caching can sometimes lead to situations with outdated data (that may happen when the data has changed in the database, but the cache hasn't been updated yet)
* caching logic can make system design more complex – that leads to complicate debugging and testing process
* resource consumption and possible problems with usage of system memory in memory-constrained environments
* cache invalidation logic (when and how to invalidate or refresh a cache) can be a challenging task

When to consider applying caching?

In the cases where:

* reading data is *time-consuming* in comparison with cache access time (due to multiple layers in the way to the db)
* the data reads are much more frequent than writes (if data is continually changing, implementing caches might become more complex and less beneficial, or it will be necessary to update the cache more often)
* same pieces of data are requested repeatedly and may be replaced with cached data
* there is a necessity of reducing the load on the db or other services that might have higher costs or rate limiting
* some parts of the app can use *slightly* outdated data without compromising overall functionality and help with heavy traffic.

1. What are the differences between In-memory, Distributed or Request caching options?

**In-Memory Caching**: This kind of caching stores data directly within the application's memory. That means the data is stored directly in the RAM of the machine where the application is running, and it is very fast to access. However, that cache depends on the lifecycle of the server (it’s data will be lost if the server goes down) and is *locally available* to the server where it has been created, so it cannot be accessed from any other servers of the system.

**Distributed Caching**: This method resolves some of the issues with in-memory caching in a multi-server environment – this type of cache presumes the data is stored in a centralized server or cluster of servers, and that data can be accessed by all the other servers of the app. These caches can be located in-memory or on disk depending on the technology used. Also, data persistence across restarts can be achieved using Distributed Caching.

**Request Caching**: This approach is used to cache data for the duration of an HTTP request. This can be useful when there is some data that may be re-utilized multiple times within the same request but isn't needed beyond that. This data is typically stored within the HTTP context of the particular request and doesn't persist beyond it. The cached data isn't shared across different requests, users or servers.

1. What does ‘session affinity’ and ‘thread affinity’ mean? When do we have to consider session affinity?

Session Affinity. That is about the server load balancing scheme where all the requests from a client during a session get sent to the same server. This ensures all requests from a user during a single session are handled by the same server, thus maintaining session data – that might be useful in situations where the application stores session-specific data in the server's memory that might not be available if the client's requests were routed to other servers.

Session affinity becomes important when there is a case of dealing with applications that store user session data server-side. Without that feature, if a user makes a couple of requests to separate servers, those may face with lack of access to the session data of the other server, leading to problems.

Thread Affinity. This refers to the situation where certain threads are designated to execute specific tasks. For example, certain resources (like UI in many desktop applications) require that all operations be performed by a specific (UI) thread, so operations that involve these resources are required to execute on the specific thread. *This is not much of a concern in web-based applications.*

Thread affinity isn't usually a concern in a typical ASP.NET Core application because every incoming HTTP request is processed in a separate worker thread from the thread pool, and the application should not be maintaining any state information specific to a given thread. But in some cases like desktop applications or some third-party libraries, it’s necessary to consider thread affinity.

1. What are the race conditions and deadlocks? Do they possible in a single threaded application?

A race condition happens when two or more operations must execute in the correct order, but the program has not been designed to guarantee this order. It's called a 'race' because the operations are racing to influence the other. So there can be a situation of reading and writing of some data: if those operations don’t happen as one atomic operation, then writing could be made from another process, leading to an unexpected behavior.

A deadlock is a state in which each member of a group is waiting for another member, including itself, to do something. That case often happens with shared resources, with two or more threads unable to proceed because each is holding a resource that the other threads are trying to lock. Looks like a traffic deadlock from a real world, where no car can move because each one is waiting for the other to give way.

Race conditions usually exist in multithreaded (or multiprocess) applications. However, it's also possible for a race condition to occur in a single-threaded environment. This can happen with asynchronous programming where the order of operations is not guaranteed, or callbacks/interrupts are used.

On the other hand, deadlocks exist in situations where there are two or more threads or processes. That happens because of a deadlock requires different threads to hold and need resources that the other controls. It's hard to have this situation in a single-threaded environment as there’s inherently only one block of execution and hence, no *concurrency* for resources. But a sort of “deadlock” could occur in a single-threaded environment if a recursive lock or reentrant lock is not handled properly.

1. Why is it not safe to use static constructors/fields when your code is running in a multithreaded application?

By design, static fields are shared across all the instances of a class and are initialized only once which means if multiple threads are accessing and changing the static fields simultaneously, this can lead to race conditions.

The common language runtime (CLR) controls the execution of the static constructors, ensuring they only run once per AppDomain, and typically run before any instance of the class is created. Using static constructors for initializing shared data could result in a situation where the constructor has not finished initializing but is read by another thread, leading to unpredictable outcomes.

Static fields are shared across all instances, so they are vulnerable to race conditions unless thread synchronization (constructs like locks, Mutex, Semaphore, etc) and some thread usage best practices are used.

Using static fields for read-only, immutable data or as constants can be thread-safe. At the same time, using thread-static fields ([ThreadStatic] attribute or ThreadLocal<T>), where each thread will have its own instance, can also be a thread-safe use of static fields.

7.      What objects and features .NET proposes to solve race conditions and deadlocks?

.NET provides several synchronization primitives and features to help handle and prevent race conditions and deadlocks in multithreaded applications. Some of them are:

Lock keyword - This keyword marks a statement block as a **critical section** by obtaining the mutual-exclusion lock for a given object, executing a statement block, and then releasing the lock, that means only one thread can execute the code block at a time.

Monitor - Similar to lock, Monitor provides a mechanism that synchronizes access to objects with methods like Enter, Exit, Wait, Pulse, and more.

Mutex - **Mutex** is similar to a lock but it can work across multiple processes, making it useful in scenarios where inter-process synchronization is required.

Semaphore - A **semaphoring** is a thread synchronization construct that can be used to protect a certain number of execution instances, unlike lock, Monitor, and Mutex which only guard one critical section.

ReaderWriterLock and ReaderWriterLockSlim – the constructions that allow multiple threads to read data simultaneously but provide an lock for writing – that may be useful in case of many read operations with less write operations.

Volatile keyword - This keyword indicates that a field can be modified in the program by the hardware. It is usually used as a protective measure in multi-threaded environments to make sure that all threads always observe the most recent write operation.

**Interlocked** class - This class provides atomic operations like Increment or CompareExchange for variables that are shared across multiple threads.

Thread.MemoryBarrier method – it inserts a memory barrier that prevents the CPU from reordering read/write operations across the barrier.

Concurrent Collections - ConcurrentStack, ConcurrentQueue, ConcurrentDictionary, BlockingCollection, etc. They are thread-safe, so can safely be accessed and modified by multiple threads at the same time.

async/await - the proper use of async and await keywords can improve the performance and responsiveness of the application and generally reduce the chance of deadlocks.

TPL Dataflow Library – a library designed to help with building concurrent and parallel data processing systems, including features for buffering, transformation, propagation and etc, to handle deadlock and race conditions.

**Ticketing Domain – Unit and Integration testing**

1. What are the benefits and drawbacks of unit tests?

The benefits:

* code quality insurance (unit tests allow to ensure the code works as intended and meets the requirements, so any changes breaking a feature may be identified quickly),
* quick code checks (if changes broken a feature or not),
* help with code design (mainly with via Test Driven Development (TDD),
* may be a part of documentation for the code and features, showing how a function should work and be used.

The drawbacks:

* require extra time for writing new tests and support of the existing ones if the requirements were updated,
* reflect only the individual components checks, but not the check of the whole complex. “All passed unit tests don’t mean there are no integration bugs”.
* don't fix/reflect all the bugs of the components, especially the exceptions occurring while interaction between components – so unit testing should not be the only form of testing.

1. What are the benefits and drawbacks of integration tests?

The benefits:

* simulate real usage scenarios,
* allow check the work of several interacting components of the system,
* allow check the interaction with the database using real network,
* increase the code coverage together with unit tests implemented

The drawbacks:

* require extra time for implementation (because of the complexity),
* may take more time for execution (as they test real components like database, network, etc),
* difficult rollbacks in cases of failed integration tests,
* integration tests may be instable (because of different factors like race conditions, timing issues, reliance on external resources (like a database or network), randomness in test data or system state, hardware variability, etc).

1. What are benefits and drawbacks of end-to-end tests?

The benefits:

* simulate real usage scenarios & workflows,
* sometimes can identify the bugs that were not found on the unit testing and the integration testing stages,
* reflect real user experience, so team is more confident in the system’s health,
* check the interaction with external systems (higher level of integration testing)

The drawbacks:

* increased complexity (require deep understanding of the whole system)
* require extra time for implementation & support (because of the complexity),
* may take more time for execution than integration tests do (as e2e tests can check interaction with external systems),
* may be instable (because of testing several components and external systems).
* increased debugging complexity,
* usually require a separate prod-like testing environment (what increases the costs and require extra maintenance)

1. When/why you would do database integration tests?

* when it’s necessary to verify that the application can execute CRUD operations over the database correctly,
* when it’s necessary to check the correctness of the transactions executed (and/or rolled back) over multiple tables (including cascading operations),
* when the efficiency of the interaction with the database should be tested,
* when testing the workflows after the migration held,
* when real database operations should be tested (calling the stored procedures, complex queries, etc)

1. How testing trophy differs from testing pyramid model?

The key difference between those models is their approach towards testing: the Pyramid advises heavy reliance on unit tests, while the Trophy encourages a more balanced approach, emphasizes the significance of integration testing and static analysis (a testing methodology that implies examination of code with identifying issues related to security, performance, design, coding style, etc – without the execution of the code itself).

1. What code coverage metrics do you know? What metric would you use?

Function coverage (each method or function has been executed at least once),

Statement coverage (the amount of the executable statements in the source code that have been run through),

Branch coverage (every possible branch from each decision point, including all the if-else statements, has been executed),

Condition coverage: (evaluates the true and false outcomes of Boolean expressions),

Line coverage: (the percentage of lines that have been executed in the source code),

Path coverage: (all possible paths through the code (combinations of branches) have been tested).

The choice depends on the needs, but I would use Statement and Branch coverage metrics (in some way they can guarantee a wide range of the program's code is examined, including everything from single lines of code to the overall logic).

1. What is practically reasonable percent of code coverage?

The average reasonable percent of code coverage is at least 70-80% - that value can ensure critical code parts are covered.

**Ticketing Domain – Asynchronous ASP.NET Core APIs**

1. What are the benefits and drawbacks of async programming?

The main benefit is the execution speed due to the fact there’s no need to wait for tasks to complete. This way responsiveness and high performance may be achieved – program remains responsive during heavy operations held asynchronously in background (the main tread is not blocked by time-consuming tasks and still able to manage other tasks efficiently). Async programming introduces the ability to perform multiple tasks concurrently. can be executed concurrently. Another benefit is improved end user experience: in applications with UI treads (web, mobile) end users don’t face with freezes of UI.

At the same time, there are a couple of drawbacks: increased complexity (understanding of tasks chains, execution order, debugging), unusual cases (like deadlocks, etc.), difficult error handling (exceptions can occur during unexpected behavior), not efficient for small tasks (in those cases asynchrony/parallelism can lead to increased execution time in comparison with synchronous variants – there is a point in a chart “execution time to amount of data” where sync/async processing reach same execution time, starting from which async becomes faster than its sync variant with increasing the data amount).

It's also important to remember that not all operations benefit from being asynchronous – async can be considered in case of I/O-bound tasks (database queries, HTTP requests, File I/O operations, etc), CPU-bound tasks (e.g. massive heavy computations) are not suitable for async (due to they don’t involve waiting for external resources)

1. How to make ASP.NET controller action support async flow?

The return type of the method should be a Task or generic Task (e.g. *Task<IActionResult>*). The method should be marked with *async* keyword. Use *await* in the method’s body on the awaitable (long-running) calls.

1. How does async flow influences ASP.NET request executions (life cycle)?

Async in ASP.NET Core allows to handle multiple requests efficiently without blocking threads. As the framework has the middleware pipeline, the request is processed in the configured order, but async middleware can return the control back to the server while waiting for external resources (database, API, etc) – for example this way routing, authentication and logging are handled. This way scalability may be achieved (multiple requests handling).

1. List at least 5 tips on ASP.NET API performance best practices?

Use asynchronous database operations (e.g. with EF Core – follow EF Core best practices, e.g. no-tracking queries).

Use pagination to chunk large data collections.

Avoid sync read/write on HttpRequest/HttpResponse body (that can lead to sync over async because Kestrel does NOT support synchronous reads).

Avoid IHttpContextAccessor.HttpContext stored in a field.

Do not access HttpContext from multiple threads.

Use Caching.

1. Vertical vs Horizontal scalability. Where to use each?

Vertical means increasing power capabilities of existing machines to meet demand (e.g. upgrading CPU, memory, storage). This type of scalability is preferred in cases of predictable growth of demand, easy for quick prototyping (there is only one unit that doesn’t need to be coordinated).

Horizontal means adding nodes (machines) to the infrastructure to handle increased demands. This is an analogy of real spread of work between employees (same to nodes), so the load and traffic may be distributed effectively. But here it’s important to remember about correct managing the nodes and their communication.

1. Explain why the PUT method was suggested for the book action on the order

There are three ways (HTTP verbs) for requesting updates of the resources: PUT, POST and PATCH.

POST is commonly used to create new resources. Usually returns the updated state of the entity (e.g. with the increased amount of something).

PATCH allows to update an existing resource (but unlike PUT which replaces the entire resource, this action will modify only specific fields or attributes), that’s why PATCH is more complex due to handling specific fields updates. This way PUT is preferred to update all seat states in a cart, but PATCH is for individual seats states updates.

PUT is used to modify (update) an existing resource completely, it’s idempotent in comparison with POST (repeated POST requests will create several entities, when the results of several PUT actions will be the same).

**Ticketing Domain – Persistence Level (SQL, NoSQL) + DAL**

1. What are your steps to start designing database?

To start designing database, it’s necessary to

- determine the purpose of the database: the kind information to be stored, its consumers (people who will use that information) and the way they will interact with it

- find and organize the information required: gather all of the types of information you might want to record in the database, create a description of the data type, size, and format

- start designing with division the information into subjects (or entities), each one then becomes a table.

1. When can we say that our database is modeled correctly?

A modeled-correctly database divides the information into subject-based tables (it helps to reduce redundant data), provides with the necessary details to gather the information from various tables as required, aids in maintaining and assuring the precision and consistency of the stored data and accommodates the stored data processing and reporting needs.

1. What is a Data Access Layer (DAL), and how does it simplify database interactions?

A Data Access Layer (DAL) is a part of the software application (module, project or any else), which is responsible for interaction with the database. In a three-tier layered application together with Business Logic (BL) and Presentation layers, forms a well-supported modular system, each part of which has its own responsibilities. DAL provides the simplified and consistent methods for accessing, retrieving and manipulating the stored data.

The simplification of the interactions is about:

- encapsulating of details of operations, leaving only a single interface

- single interface provision allows the BL layer’s services to be independent on the type of database used, so any changes to the DB structure (adding a new table, changing a data type) don’t affect the business logic. Same to complete change of DB (e.g. from MySQL to PostgreSQL)

- increased security to validate queries for SQL injections.

1. You need to implement a new service for a customer. How would you select database (SQL or NoSQL)?

The choice depends on

- budget bounds

- the way data should be structured (NoSQL will be more flexible and will handle any structural changes of data or storing the unstructured one)

- the speed/consistency needs (SQL databases follow ACID principles and its data is always consistent, but NoSQL doesn’t follow with having a speed and simplified scalability)

- needs to handle complex queries (SQL databases have powerful querying capabilities, when NoSQL are not good at dealing with complex queries, but cope with quick processing of large amounts of high-velocity, variable data).

**Ticketing Domain – Introduction**

1. What diagram types do you have on your current project?

There is a component diagram and a state diagram on my current project.

1. What are the pros and cons of UML diagrams?   
   – learning of UML may be a long process (due to variety of standard details and diagram types)

– usually take much time for creation  
– take time for post-creation support (keeping them up to date)   
– it’s important to keep the diagrams simple – that influences on their readability and understanding

+ standardization

+ don’t require technical knowledge to be reviewed by customers, clear for clients, designers and developers

+ allow to decompose complex systems into smaller pieces

1. What is the difference between a structural diagram and a behavioral diagram in UML?

As their names say, structural diagram shows the structure of the system components and their relationship & interaction.

A behavioral diagram describes the way of processes held between the components, how they work over time, the messages and events between the components and the way they are handled.

1. Name the relationship types in a use case diagram

Association, Extend, Include, Use case or actor generalization

1. What is the sequence diagram?

A type of UML diagram that shows the way and the order of objects operate with one another over time. The objects involved in the sequence are represented as a vertical dotted “lifelines”. The interaction between those lifelines is shown with horizontal arrows in the same order (from up to down) the actions are performed