**Software Architecture Interview**

**Q1: What Is Load Balancing?**  Related To: Load Balancing

Load balancing is simple technique for distributing workloads across multiple machines or clusters. The most common and simple load balancing algorithm is Round Robin. In this type of load balancing the request is divided in circular order ensuring all machines get equal number of requests and no single machine is overloaded or underloaded.

**The Purpose of load balancing is to**

* Optimize resource usage (avoid overload and under-load of any machines)
* Achieve Maximum Throughput
* Minimize response time

**Most common load balancing techniques in web based applications are**

* Round robin
* Session affinity or sticky session
* IP Address affinity

**2: What Is CAP Theorem?** Related To: CAP Theorem

**The CAP Theorem for distributed computing** was published by Eric Brewer. This states that it is not possible for a distributed computer system to simultaneously provide all three of the following guarantees:

1. **Consistency** (all nodes see the same data even at the same time with concurrent updates )
2. **Availability** (a guarantee that every request receives a response about whether it was successful or failed)
3. **Partition tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)

The CAP acronym corresponds to these three guarantees. This theorem has created the base for modern distributed computing approaches. Worlds most high volume traffic companies (e.g. Amazon, Google, Facebook) use this as basis for deciding their application architecture. It's important to understand that only two of these three conditions can be guaranteed to be met by a system.

**3: Define Microservice Architecture** Related To: Microservices

Microservices, aka Microservice Architecture, is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

**4: Why use WebSocket over HTTP?** Related To: WebSockets

A WebSocket is a continuous connection between client and server. That continuous connection allows the following:

1. Data can be sent from server to client at any time, without the client even requesting it. This is often called server-push and is very valuable for applications where the client needs to know fairly quickly when something happens on the server (like a new chat messages has been received or a new price has been udpated). A client cannot be pushed data over http. The client would have to regularly poll by making an http request every few seconds in order to get timely new data. Client polling is not efficient.
2. Data can be sent either way very efficiently. Because the connection is already established and a webSocket data frame is very efficiently organized, one can send data a lot more efficiently that via an HTTP request that necessarily contains headers, cookies, etc...

**5: What do you mean by lower latency interaction?** Related To: WebSockets

**Low latency means** that there is very little delay between the time you request something and the time you get a response. As it applies to webSockets, it just means that data can be sent quicker (particularly over slow links) because the connection has already been established so no extra packet roundtrips are required to establish the TCP connection.

**6: What Is Scalability? Related To: Scalability**

Scalability is the ability of a system, network, or process to handle a growing amount of load by adding more resources. The adding of resource can be done in two ways

* **Scaling Up**

This involves adding more resources to the existing nodes. For example, adding more RAM, Storage or processing power.

* **Scaling Out**

This involves adding more nodes to support more users.

Any of the approaches can be used for scaling up/out a application, however the cost of adding resources (per user) may change as the volume increases. If we add resources to the system It should increase the ability of application to take more load in a proportional manner of added resources.

An ideal application should be able to serve high level of load in less resources. However, in practical, linearly scalable system may be the best option achievable. Poorly designed applications may have really high cost on scaling up/out since it will require more resources/user as the load increases.

**7 Why Do You Need Clustering?**

Clustering is needed for achieving high availability for a server software. The main purpose of clustering is to achieve 100% availability or a zero down time in service. A typical server software can be running on one computer machine and it can serve as long as there is no hardware failure or some other failure. By creating a cluster of more than one machine, we can reduce the chances of our service going un-available in case one of the machine fails.

Doing clustering does not always guarantee that service will be 100% available since there can still be a chance that all the machine in a cluster fail at the same time. However it in not very likely in case you have many machines and they are located at different location or supported by their own resources.

**8: What Is A Cluster?**

A cluster is group of computer machines that can individually run a software. Clusters are typically utilized to achieve high availability for a server software. Clustering is used in many types of servers for high availability.

**App Server Cluster**

An app server cluster is group of machines that can run a application server that can be reliably utilized with a minimum of down-time.

**Database Server Cluster**

An database server cluster is group of machines that can run a database server that can be reliably utilized with a minimum of down-time.

**9 What is Domain Driven Design?** Related To: DDD

Domain Driven Design is a methodology and process prescription for the development of complex systems whose focus is mapping activities, tasks, events, and data within a problem domain into the technology artifacts of a solution domain.

It is all about trying to make your software a model of a real-world system or process.

**10: What defines a software architect?**

An **architect** is the captain of the ship, making the decisions that cross multiple areas of concern (navigation, engineering, and so on), taking final responsibility for the overall health of the ship and its crew (project and its members), able to step into any station to perform those duties as the need arises (write code for any part of the project should they lose a member). He has to be familiar with the problem domain, the technology involved, and keep an eye out on new technologies that might make the project easier or answer new customers' feature requests.

**11: What is meant by the KISS principle?**

KISS, a backronym for "keep it simple, stupid", is a design principle noted by the U.S. Navy in 1960. The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design, and that unnecessary complexity should be avoided.

**12: Why is it a good idea for “lower” application layers not to be aware of “higher” ones?** Related To: Layering & Middleware

The fundamental motivation is this:

You want to be able to rip an entire layer out and substitute a completely different (rewritten) one, and NOBODY SHOULD (BE ABLE TO) NOTICE THE DIFFERENCE.

The most obvious example is ripping the bottom layer out and substituting a different one. This is what you do when you develop the upper layer(s) against a simulation of the hardware, and then substitute in the real hardware.

Also layers, modules, indeed architecture itself, are means of making computer programs easier to understand by humans.

**13: What is Test Driven Development?** Related To: Agile & Scrum

Test Driven Development (TDD) is also known as test-driven design. In this method, developer:

1. first writes an automated test case which describes new function or improvement and
2. then creates small codes to pass that test, and
3. later re-factors the new code to meet the acceptable standards.

**14: What does the expression “Fail Early” mean, and when would you want to do so?**

Essentially, fail fast (a.k.a. fail early) is to code your software such that, when there is a problem, the software fails as soon as and as visibly as possible, rather than trying to proceed in a possibly unstable state.

Fail Fast approach won’t reduce the overall number of bugs, at least not at first, but it’ll make most defects much easier to find.

**15: What does program to interfaces, not implementations mean?**

Related To: Design Patterns

**Coding against interface** means, the client code always holds an Interface object which is supplied by a factory.

Any instance returned by the factory would be of type Interface which any factory candidate class must have implemented. This way the client program is not worried about implementation and the interface signature determines what all operations can be done.

This approach can be used to change the behavior of a program at run-time. It also helps you to write far better programs from the maintenance point of view.

**16: What is Elasticity (in contrast to Scalability)?** Related To: Scalability

**Elasticity** means that the throughput of a system scales up or down automatically to meet varying demand as resource is proportionally added or removed. The system needs to be scalable to allow it to benefit from the dynamic addition, or removal, of resources at runtime. Elasticity therefore builds upon scalability and expands on it by adding the notion of automatic resource management.

**17: What is Back-Pressure?** Related To: Availability & Reliability

When one component is struggling to keep-up, the system as a whole needs to respond in a sensible way. It is unacceptable for the component under stress to fail catastrophically or to drop messages in an uncontrolled fashion. Since it can’t cope and it can’t fail it should communicate the fact that it is under stress to upstream components and so get them to reduce the load.

This back-pressure is an important feedback mechanism that allows systems to gracefully respond to load rather than collapse under it. The back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load.

**18: WebSockets vs Rest API for real time data? Which to choose?**

Related To: API Design, WebSockets, REST & RESTful

**Problem**

I need to constantly access a server to get real time data of financial instruments. The price is constantly changing so I need to request new prices every 0.5 seconds. Which kind if API would you recommend?

**Answer**

The most efficient operation for what you're describing would be to use a webSocket connection between client and server and have the server send updated price information directly to the client over the webSocket ONLY when the price changes by some meaningful amount or when some minimum amount of time has elapsed and the price has changed.

Here's a comparison of the networking operations involved in sending a price change over an already open webSocket vs. making a REST call.

**webSocket**

1. Server sees that a price has changed and immediately sends a message to each client.
2. Client receives the message about new price.

**Rest/Ajax**

1. Client sets up a polling interval
2. Upon next polling interval trigger, client creates socket connection to server
3. Server receives request to open new socket
4. When connection is made with the server, client sends request for new pricing info to server
5. Server receives request for new pricing info and sends reply with new data (if any).
6. Client receives new pricing data
7. Client closes socket
8. Server receives socket close

As you can see there's a lot more going on in the Rest/Ajax call from a networking point of view because a new connection has to be established for every new call whereas the webSocket uses an already open call. In addition, in the webSocket cases, the server just sends the client new data when new data is available - the client doens't have to regularly request it.

A webSocket can also be faster and easier on your networking infrastructure simply because fewer network operations are involved to simply send a packet over an already open webSocket connection versus creating a new connection for each REST/Ajax call, sending new data, then closing the connection. How much of a difference/improvement this makes in your particular application would be something you'd have to measure to really know.

**19: What is the difference between Monolithic, SOA and Microservices Architecture?** Related To: Microservices, SOA

* **Monolithic Architecture** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
* **A Service-Oriented Architecture** is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
* **Microservice Architecture** is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

**20: What Is Session Replication?**

**Session replication** is used in application server clusters to achieve session failover. A user session is replicated to other machines of a cluster, every time the session data changes. If a machine fails, the load balancer can simply send incoming requests to another server in the cluster. The user can be sent to any server in the cluster since all machines in a cluster have copy of the session.

Session replication may allow your application to have session failover but it may require you to have extra cost in terms of memory and network bandwidth.

**21: What Is Middle Tier Clustering?** Related To: Layering & Middleware

Middle tier clustering is just a cluster that is used for service the middle tier in a application. This is popular since many clients may be using middle tier and a lot of heavy load may also be served by middle tier that requires it be to highly available.

Failure of middle tier can cause multiple clients and systems to fail, therefore its one of the approaches to do clustering at the middle tier of a application. In general any application that has a business logic that can be shared across multiple client can use a middle tier cluster for high availability.

**22: How Do you update a live heavy traffic site with minimum or Zero Down Time?**

Related To: Availability & Reliability, DevOps

Deploying a newer version of a live website can be a challenging task specially when a website has high traffic. Any downtime is going to affect the users. There are a few best practices that we can follow:

**Before deploying on Production:**

* Thoroughly test the new changes and ensure it working in a test environment which is almost identical to production system.
* If possible do automation of test cases as much as possible.
* Create a automated sanity testing script (also called as smoke test) that can be run on production (without affecting real data). These are typically readonly type of test cases. However depending on your application needs you can add more cases to this. Make sure it can be run quickly by keeping it short.
* Create scripts for all manual tasks(if possible), avoiding any hand typing mistakes during day of deployment.
* Test the script to make sure they work on a non-production environment.
* Keep the build artifacts ready. e.g application deployment files, database scripts, config files etc.
* Create a checklist of things to do on day of deployment.
* Rehearse. Deploy in a non-prod environment is almost identical to production. Try this with production data volumes(if possible). Make a note of time required for your tasks so you can plan accordingly.

**When doing deploying on a production environment:**

* Use Green-Blue deployment technique to reduce down-time risk
* Keep backup of current site/data to be able to rollback
* Use sanity test cases before doing a lot of in depth testing

**23: What Is ACID Property Of A System?** Related To: Databases

ACID is a acronym which is commonly used to define the properties of a relational database system, it stand for following terms

* **Atomicity** - This property guarantees that if one part of the transaction fails, the entire transaction will fail, and the database state will be left unchanged.
* **Consistency** - This property ensures that any transaction will bring the database from one valid state to another.
* **Isolation** - This property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially.
* **Durable** - means that once a transaction has been committed, it will remain so, even in the event of power loss.

**24: What Is Sticky Session Load Balancing? What Do You Mean By "Session Affinity"?**

Related To: Load Balancing

Sticky session or a session affinity technique is another popular load balancing technique that requires a user session to be always served by an allocated machine.

In a load balanced server application where user information is stored in session it will be required to keep the session data available to all machines. This can be avoided by always serving a particular user session request from one machine. The machine is associated with a session as soon as the session is created. All the requests in a particular session are always redirected to the associated machine. This ensures the user data is only at one machine and load is also shared.

This is typically done by using SessionId cookie. The cookie is sent to the client for the first request and every subsequent request by client must be containing that same cookie to identify the session.

**What Are The Issues With Sticky Session?**

There are few issues that you may face with this approach

* The client browser may not support cookies, and your load balancer will not be able to identify if a request belongs to a session. This may cause strange behavior for the users who use no cookie based browsers.
* In case one of the machine fails or goes down, the user information (served by that machine) will be lost and there will be no way to recover user session.

**25: What Do You Mean By High Availability (HA)?**  Related To: Availability & Reliability

Availability means the ability of the application user to access the system, If a user cannot access the application, it is assumed unavailable. High Availability means the application will be available, without interruption. Using redundant server nodes with clustering is a common way to achieve higher level of availability in web applications.

Availability is commonly expressed as a percentage of uptime in a given year.

**26: What does it mean "System Shall Be Resilient"?** Related To: Availability & Reliability

System is **Resilient** if it stays **responsive** in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure.

Resilience is achieved by:

* replication,
* containment,
* isolation and
* delegation.

Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

**27: What is a Model in DDD?** Related To: DDD

A model is a useful approximation to the problem at hand.

An **Employee** class is not a real employee. It models a real employee. We know that the model does not capture everything about real employees, and that's not the point of it. It's only meant to capture what we are interested in for the current context.

Different domains may be interested in different ways to model the same thing. For example, the salary department and the human resources department may model employees in different ways.

**28: What is Domain in DDD?** Related To: DDD

In order to create good software, you have to know what that software is all about. You cannot create a banking software system unless you have a good understanding of what banking is all about, one must understand the domain of banking.

**Domain** is the field for which a system is built. Airport management, insurance sales, coffee shops, orbital flight, you name it.

It's not unusual for an application to span several different domains. For example, an online retail system might be working in the domains of shipping (picking appropriate ways to deliver, depending on items and destination), pricing (including promotions and user-specific pricing by, say, location), and recommendations (calculating related products by purchase history).

**29: Explain the Single Responsibility Principle (SRP)?**

Single responsibility is the concept of a Class doing one specific thing (responsibility) and not trying to do more than it should, which is also referred to as High Cohesion.

Classes don't often start out with Low Cohesion, but typically after several releases and different developers adding onto them, suddenly you'll notice that it became a monster or God class as some call it. So the class should be refactored.

**30: What is difference between Fault Tolerance and Fault Resilience?**

* Fault tolerance: User does not see any impact except for some delay during which failover occurs.
* Fault resilience: Failure is observed. But rest of system continues to function normally.

**31: What is the difference between Concurrency and Parallelism?**

Related To: Concurrency

**Concurrency** is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

**Parallelism** is when tasks literally run at the same time, e.g., on a multicore processor.

For instance a bartender is able to look after several customers while he can only prepare one beverage at a time. So he can provide concurrency without parallelism.

**32: What is the difference between DTOs and ViewModels in DDD?** Related To: DDD

* The canonical definition of a DTO is the data shape of an object without any behavior. Generally DTOs are used to ship data from one layer to another layer across process boundries.
* ViewModels are the model of the view. ViewModels typically are full or partial data from one or more objects (or DTOs) plus any additional members specific to the view's behavior (methods that can be executed by the view, properties to indicate how toggle view elements etc...). In the MVVM pattern the ViewModel is used to isolate the Model from the View.

**33: What Is Load Balancing Fail Over?** Related To: Load Balancing

Fail over means switching to another machine when one of the machine fails. Fail over is a important technique in achieving high availability. Typically a load balancer is configured to fail over to another machine when the main machine fails.

To achieve least down time, most load balancer support a feature of heart beat check. This ensures that target machine is responding. As soon as a hear beat signal fails, load balancer stops sending request to that machine and redirects to other machines or cluster.

**34: What are the DRY and DIE principles?**

In software engineering, **Don't Repeat Yourself (DRY)** or **Duplication is Evil (DIE)** is a principle of software development.

**35: What does SOLID stand for? What are its principles?**

S.O.L.I.D is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin.

* **S - Single-responsiblity principle**. A class should have one and only one reason to change, meaning that a class should have only one job.
* **O - Open-closed principle.** Objects or entities should be open for extension, but closed for modification.
* **L - Liskov substitution principle**. Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
* **I - Interface segregation principle.** A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
* **D - Dependency Inversion Principle**. Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

**36: Is it better to return NULL or empty values from functions/methods where the return value is not present?**

Returning null is usually the best idea if you intend to indicate that no data is available.

An empty object implies data has been returned, whereas returning null clearly indicates that nothing has been returned.

Additionally, returning a null will result in a null exception if you attempt to access members in the object, which can be useful for highlighting buggy code - attempting to access a member of nothing makes no sense. Accessing members of an empty object will not fail meaning bugs can go undiscovered.

**37: "People who like this also like... ". How would you implement this feature in an e-commerce shop?**

This is open-ended question. Reference to your experience to provide a relevant answer.

**38: How can you keep one copy of your utility code and let multiple consumer components use and deploy it?**

Once you start growing and have different components on different servers which consumes similar utilities, you should start managing the dependencies .

There is a tool for that, it's called npm or nuget. Start by wrapping 3rd party utility packages with your own code to make it easily replaceable in the future and publish your own code as private npm package. Now, all your code base can import that code and benefit free dependency management tool. It's possible to publish npm packages for your own private use without sharing it publicly using private modules, private registry or local npm packages

**39: Name some Performance Testing best practices**

Related To: Software Testing

* Test as early as possible in development.
* Conduct multiple performance tests to ensure consistent findings and determine metrics averages.
* Test the individual software units separately as well as together
* Baseline measurements provide a starting point for determining success or failure
* Performance tests are best conducted in test environments that are as close to the production systems as possible
* Isolate the performance test environment from the environment used for quality assurance testing
* Keep the test environment as consistent as possible
* Calculating averages will deliver actionable metrics. There is value in tracking outliers also. Those extreme measurements could reveal possible failures.

**40: Name some Performance Testing metrics to measure**

Related To: Software Testing

1. **Response time** - Total time to send a request and get a response.
2. **Wait time** - Also known as average latency, this tells developers how long it takes to receive the first byte after a request is sent.
3. **Average load time** - The average amount of time it takes to deliver every request is a major indicator of quality from a user’s perspective.
4. **Peak response time** - This is the measurement of the longest amount of time it takes to fulfill a request. A peak response time that is significantly longer than average may indicate an anomaly that will create problems.
5. **Error rate** - This calculation is a percentage of requests resulting in errors compared to all requests. These errors usually occur when the load exceeds capacity.
6. **Concurrent users** - This the most common measure of load — how many active users at any point. Also known as load size.
7. **Requests per second** - How many requests are handled.
8. **Transactions passed/failed** - A measurement of the total numbers of successful or unsuccessful requests.
9. **Throughput** - Measured by kilobytes per second, throughput shows the amount of bandwidth used during the test.
10. **CPU utilization** - How much time the CPU needs to process requests. Memory utilization - How much memory is needed to process the request.

**41: Is Unit Of Work equals Transaction? Or it is more than that?** Related To: Design Patterns

A **UnitOfWork** is a business transaction. Not necessarily a technical transaction (db transaction) but often tied to technical transactions.

**In the Enterprise Application Patterns it is defined as**

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

**A database transaction** with a number of SQL statements in between is arguably also a Unit of Work. However, the key difference, is that the unit of work, as defined in the pattern, has abstracted that level of detail (how changes are written and the storage type) to an object level.

**42: In OOP, what is the difference between the Repository Pattern and a Service Layer?**

Related to : Design Patterns, Web API

* **Repository Layer** gives you additional level of abstraction over data access.
* **Service Layer** exposes business logic, which uses repository or a set of repositories as UnitOfWork

In ASP.NET MVC + EF + SQL Server, I have this flow of communication:

* Views <- Controllers -> Service layer -> Repository layer -> EF -> SQL Server
* Service layer -> Repository layer -> EF This part operates on models.
* Views <- Controllers -> Service layer This part operates on view models.

**43: What is Bad Design?**

If a system exhibits any or all of the following three traits then we have identified bad design:

* the system is **rigid**: it’s hard to change a part of the system without affecting too many other parts of the system
* the system is **fragile**: when making a change, unexpected parts of the system break
* the system or component is **immobile**: it is hard to reuse it in another application because it cannot be disentangled from the current application.

**44: What is the difference between Behavior-Driven Development (BDD) vs Domain-Driven Design (DDD)?**

Related To: DDD

* **DDD** focuses on defining the vocabulary in that language: actors, entities, operations, ... An important part of DDD is also that the ubiquitous language can be clearly seen in the code, too, not only in communication between the implementor and the domain expert. So an extreme view of DDD is quite static: it describes the finished system as a whole.
* **BDD** focuses on defining user stories or scenarios. It is closely related to an incremental process, but it can also be viewed as static: it describes all the interactions between users and the finished system.

DDD and BDD can be applied with no overlapping: BDD user stories can be written using only UI vocabulary (buttons, labels, ...), and DDD can avoid mentioning interactions.

But ideally these two overlap and work together: the BDD stories are rich in the ubiquitous language, describing the user experience with domain concepts. And DDD makes sure the stories can be found in the code.

**45: What is the Command and Query Responsibility Segregation (CQRS) Pattern?**

Related To: DDD, Design Patterns

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations.

**CQRS** stands for **Command and Query Responsibility Segregation**, a pattern that separates **read** and **update** operations for a **data store**. CQRS separates reads and writes into different models, using **commands** to update data, and **queries** to read data.

* **Commands** should be task-based, rather than data-centric. ("Book hotel room", not "set ReservationStatus to Reserved").
* **Commands** may be placed on a queue for asynchronous processing, rather than being processed synchronously.
* **Queries** never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

**46: Name some benefits of CQRS Pattern Related To: DDD, Design Patterns**

Benefits of CQRS include:

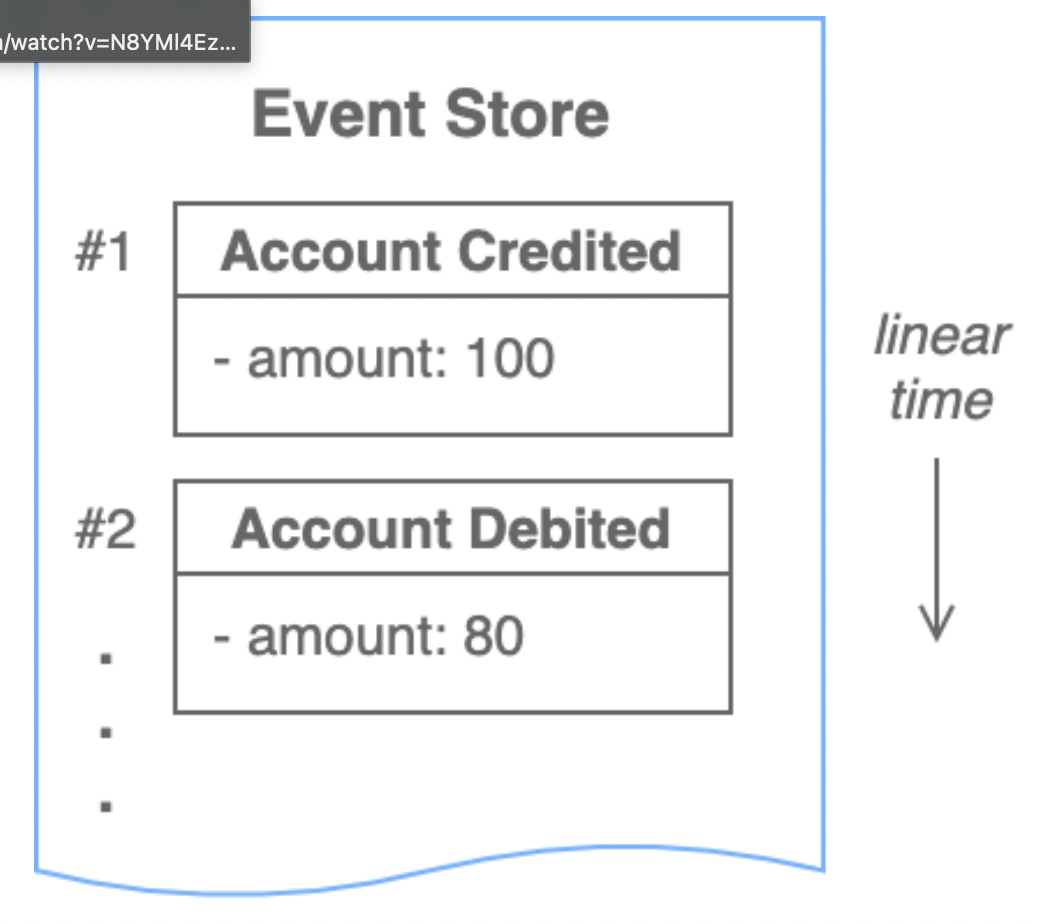
* **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
* **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
* **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
* **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
* **Simpler queries.** By storing a materialized view in the read database, the application can avoid complex joins when querying.

**47: Describe what is the Event Sourcing Pattern** Related To: DDD, Design Patterns

Event Sourcing is a pattern for the recording of state in a non-destructive way. Each change of state is appended to a log. Because the changes are non-destructive, we preserve the ability to answer queries about the state of the object at any point in its life cycle.

* Event sourcing enables traceability of changes.
* Event sourcing enables audit logs without any additional effort.
* Event sourcing makes it possible to reinterpret the past.
* Event sourcing reduces the conflict potential of simultaneously occurring changes.
* Event sourcing enables easy versioning of business logic.
* Event Sourcing is not necessary for CQRS. You can combine Event Sourcing and CQRS.

This kind of combination can lead us to a new type of CQRS. It involves modelling the state changes made by applications as an immutable sequence or log of events.



**48: Two customers add a product to the basket in the same time whose the stock was only one (1). What will you do?**

Related To: Concurrency

**What is the best practice to manage the case where two customers add in the same time a product whose the stock was only 1?**

There is no perfect answer for this question and all depends on details but you have some options:

1. As a first 'defense line' I would try to avoid such situations at all, by simply not selling out articles that low if any possible.
2. You reserve an item for the customer for a fix time say (20 minutes) after they have added it to the basket – after they time they have to recheck the stock level or start again. This is often used to ticket to events or airline seats
3. For small jobs the best way is to do a final check right before the payment, when the order is actually placed. In worst case you have to tell you customer that you where running out of stock right now and offer alternatives or discount coupon.
4. Try to fulfil both orders later - just cause you don't have stock right now, doesn't mean you can't find it in an emergency. If you can't then someone has to contact the user who lucked out and apologise.

**Side note:**

1. The solution to do the double check when adding something to the basket isn't very good. People put a lot in baskets without ever actually placing an order. So this may block this article for a certain period of time.

**49: Explain Failure in contrast to Error** Related To: Availability & Reliability

* A **failure** is an unexpected event within a service that prevents it from continuing to function normally. A failure will generally prevent responses to the current, and possibly all following, client requests.
* This is in contrast with an error, which is an expected and coded-for condition—for example an error discovered during input validation, that will be communicated to the client as part of the normal processing of the message.

Failures are unexpected and will require intervention before the system can resume at the same level of operation. This does not mean that failures are always fatal, rather that some capacity of the system will be reduced following a failure. Errors are an expected part of normal operations, are dealt with immediately and the system will continue to operate at the same capacity following an error.

**50: Provide Definition Of Location Transparency**

Location transparency enables resources to be accessed without knowledge of their physical or network location. In other words users of a distributed system should not have to be aware of where a resource is physically located.

**51: What Is Sharding?** Related To: Databases

**Sharding** is a architectural approach that distributes a single logical database system into a cluster of machines. Sharding is Horizontal partitioning design scheme. In this database design rows of a database table are stored separately, instead of splitting into columns (like in normalization and vertical partitioning). Each partition is called as a shard, which can be independently located on a separate database server or physical location.

Sharding makes a database system highly scalable. The total number of rows in each table in each database is reduced since the tables are divided and distributed into multiple servers. This reduces the index size, which generally means improved search performance. The most common approach for creating shards is by the use of consistent hashing of a unique id in application (e.g. user id).

**The downsides of sharding:**

* It requires application to be aware of the data location.
* Any addition or deletion of nodes from system will require some rebalance to be done in the system.
* If you require lot of cross node join queries then your performance will be really bad. Therefore, knowing how the data will be used for querying becomes really important.
* A wrong sharding logic may result in worse performance. Therefore make sure you shard based on the application need.

**52: Why layering your application is important? Provide some bad layering example.**

Related To: Layering & Middleware

Each component should contain 'layers' - a dedicated object for the web, logic and data access code. This not only draws a clean separation of concerns but also significantly eases mocking and testing the system.

Though this is a very common pattern, API developers tend to mix layers by passing the web layer objects (for example Express req, res) to business logic and data layers - this makes your application dependant on and accessible by Express only. App that mixes web objects with other layers can not be accessed by testing code, CRON jobs and other non-Express callers

**53: What's the difference between principles YAGNI and KISS?**

**YAGNI (You aint gona need it)** refers to over analyzing and implementing things that may or may not be needed. Sure algorithmic elegance is nice and all but most situation you dont need it. In general engineering terms you should be carefull not to include your own requirements so that you dont taint your customer needs with your own ideas that end up costing the project with little impact for the client.

**KISS (Keep it simple stupid)** refers to the fact that easy systems are easier to manage. Keeping things simple is not nesseserily less work (like YAGNI is) since it requires more knowlege to implement. They are sometimes similar but grow from different needs.

YAGNI grows from a too much future anticipation, overzealous workers if you may. KISS is a strategy that tries to counteract human tendency for design creep.

**54: What is GOD class and why should we avoid it?**

The most effective way to break applications it to create GOD classes. That are classes that keeps track of a lot of information and have several responsibilities. One code change will most likely affect other parts of the class and therefore indirectly all other classes that uses it. That in turn leads to an even bigger maintenance mess since no one dares to do any changes other than adding new functionality to it.

**55: Why should you structure your solution by components?**

Related To: Layering & Middleware

For medium sized apps and above, monoliths are really bad - having one big software with many dependencies is just hard to reason about and often leads to spaghetti code. Even smart architects — those who are skilled enough to tame the beast and 'modularize' it — spend great mental effort on design, and each change requires carefully evaluating the impact on other dependent objects.

The ultimate solution is to develop small software: divide the whole stack into self-contained components that don't share files with others, each constitutes very few files (e.g. API, service, data access, test, etc.) so that it's very easy to reason about it.

Some may call this 'microservices' architecture — it's important to understand that microservices are not a spec which you must follow, but rather a set of principles.

* Structure your solution by self-contained components is good (orders, users...)
* Group your files by technical role is bad (ie. controllers, models, helpers...)

**56: What is BASE property of a system?** Related To: Databases, NoSQL

BASE properties are the common properties of recently evolved NoSQL databases. According to CAP theorem, a BASE system does not guarantee consistency. This is a contrived acronym that is mapped to following property of a system in terms of the CAP theorem:

* **Basically available** indicates that the system is guaranteed to be available
* **Soft state** indicates that the state of the system may change over time, even without input. This is mainly due to the eventually consistent model.
* **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

**57: Explain threads to your grandparents**

This is open-ended question. Reference to your experience to provide a relevant answer.

**58: How to handle exceptions in a layered application?**

Related To: Layering & Middleware

I would stick with two basic rules:

1. Only catch exceptions that you can handle to rescue the situation. That means that you should only catch exceptions if you, by handling it, can let the application continue as (almost) expected.
2. Do not let layer-specific exceptions propagate up the call stack. create a more generic exception such as LayerException which would contain some context such as which function failed (and with which parameters) and why. I would also include the original exception as an inner exception.

Consider:

| public class UserRepository : IUserRepository  {  public IList<User> Search(string value)  {  try  {  return CreateConnectionAndACommandAndReturnAList("WHERE value=@value", Parameter.New("value", value));  }  catch (SqlException err)  {  var msg = String.Format("Ohh no! Failed to search after users with '{0}' as search string", value);  throw new DataSourceException(msg, err);  }  }  } |
| --- |

**59: Why should I isolate my domain entities from my presentation layer?**

Related To: Layering & Middleware

**Problem**

One part of domain-driven design that there doesn't seem to be a lot of detail on, is how and why you should isolate your domain model from your interface. I'm trying to convince my colleagues that this is a good practice, but I don't seem to be making much headway...

**Answer**

The problem is, as time goes on, things get added on both sides. Presentation changes, and the needs of the presentation layer evolve to include things that are completely independent of your business layer (color, for example). Meanwhile, your domain objects change over time, and if you don't have appropriate decoupling from your interface, you run the risk of screwing up your interface layer by making seemingly benign changes to your business objects.

There are cases where a DTO makes sense to use in presentaton. Let's say I want to show a drop down of Companies in my system and I need their id to bind the value to.

Well instead of loading a CompanyObject which might have references to subscriptions or who knows what else, I could send back a DTO with the name and id.

If you keep only one domain object, for use in the presentation AND domain layer, then that one object soon gets monolithic. It starts to include UI validation code, UI navigation code, and UI generation code. Then, you soon add all of the business layer methods on top of that. Now your business layer and UI are all mixed up, and all of them are messing around at the domain entity layer.

**60: What Is IP Address Affinity Technique For Load Balancing?**Related To: Load Balancing

IP address affinity is another popular way to do load balancing. In this approach, the client IP address is associated with a server node. All requests from a client IP address are served by one server node.

This approach can be really easy to implement since IP address is always available in a HTTP request header and no additional settings need to be performed. This type of load balancing can be useful if you clients are likely to have disabled cookies.

However there is a down side of this approach. If many of your users are behind a NATed IP address then all of them will end up using the same server node. This may cause uneven load on your server nodes. NATed IP address is really common, in fact anytime you are browsing from a office network its likely that you and all your coworkers are using same NATed IP address.

**61: What is actor model in context of a programming language?**

The Actor model adopts the philosophy that everything is an actor. This is similar to the everything is an object philosophy used by some object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent. The Actor model is about the semantics of message passing.

**62: Defend the monolithic architecture.**

This is open-ended question. Reference to your experience to provide a relevant answer.

**63: What is Unit test, Integration Test, Smoke test, Regression Test and what are the differences between them?**

Related To: Software Testing, Unit Testing

1. **Unit test**: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.
2. **Integration test**: Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.
3. **Smoke test (aka Sanity check)**: A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up.

* Smoke testing is both an analogy with electronics, where the first test occurs when powering up a circuit (if it smokes, it's bad!)...
* ... and, apparently, with plumbing, where a system of pipes is literally filled by smoke and then checked visually. If anything smokes, the system is leaky.

1. **Regression test:** A test that was written when a bug was fixed. It ensures that this specific bug will not occur again. The full name is "non-regression test". It can also be a test made prior to changing an application to make sure the application provides the same outcome.

**To this, I will add:**

1. **Acceptance test**: Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.
2. **System test**: Tests a system as a black box. Dependencies on other systems are often mocked or stubbed during the test (otherwise it would be more of an integration test).
3. **Pre-flight check:** Tests that are repeated in a production-like environment, to alleviate the 'builds on my machine' syndrome. Often this is realized by doing an acceptance or smoke test in a production like environment.
4. **Canary test** is an automated, non-destructive test that is run on a regular basis in a LIVE environment, such that if it ever fails, something really bad has happened. Examples might be:

* Has data that should only ever be available in DEV/TEST appeared in LIVE.
* Has a background process failed to run
* Can a user logon

**64: How do you off load work from the Database?** Related To: Databases

Here is a list of standard options.

1. **Optimize the access** to the database to only do what you need, efficiently. A good DBA can help here a lot. This is a basic step that most companies do.
2. **Cache data away from the database** using something like memcached. This is usually done at the application layer, and is highly effective. Virtually every competent website should do this.
3. More ambitiously, maintain **read-only copies of the database**, and direct queries there when possible. On the database side the necessary technology is called "replication" and the read-only copies are often also backups for failover from the main database. If you're doing a million dynamic pages per hour, odds are that you are doing this, or have thought about it.
4. Buy really, really **expensive hardware** for the database. I know that PayPal did this as of 4 years ago, and changing their architecture would have been difficult so they possibly still are.
5. **Shard the database** into multiple pieces with ranges of data. This is a very intrusive change into application design. A well-known example of a company that does this is eBay.
6. Try to use a database that **scales onto multiple machines**. Oracle RAC scales onto clusters, but doesn't let you distribute data widely. Other offerings exist that are supposed to be easier to distribute, including Microsoft's SQL Azure and FathomDB. I have not used those offerings and don't know how well they work. I suspect better than nothing, but I doubt they scale horizontally that well.
7. Relational databases generally try to provide ACID guarantees. But the CAP theorem makes it very difficult to do that in a distributed system, particularly while letting you do things like join data. Therefore people have come up with many **NoSQL alternatives** that explicitly offer weaker guarantees and avoid problematic operations in return for **fully distributed scalability.** Well-known examples of companies that use scalable NoSQL data stores include Google, Facebook and Twitter.

**65: Explain what is Cache Stampede** Related To: Caching

**A cache stampede (or cache miss storm)** is a type of cascading failure that can occur when massively parallel computingsystems with caching mechanisms come under very high load. This behaviour is sometimes also called dog-piling.

Under very heavy load, when the cached version of the page (or resource) expires, there may be sufficient concurrency in the server farm that multiple threads of execution will all attempt to render the content (or get a resource) of that page simultaneously.

To give a concrete example, assume the page in consideration takes 3 seconds to render and we have a traffic of 10 requests per second. Then, when the cached page expires, we have 30 processes simultaneously recomputing the rendering of the page and updating the cache with the rendered page.

Consider:

| function fetch(key, ttl) {  value ← cache\_read(key)  if (!value) {  value ← recompute\_value()  cache\_write(key, value, ttl)  }  return value  } |
| --- |

If the function **recompute\_value()** takes a long time and the key is accessed frequently, many processes will simultaneously call recompute\\_value() upon expiration of the cache value.

In typical web applications, the function **recompute\_value()** may query a database, access other services, or perform some complicated operation (which is why this particular computation is being cached in the first place). When the request rate is high, the database (or any other shared resource) will suffer from an overload of requests/queries, which may in turn cause a system collapse.

**66: Compare "Fail Fast" vs "Robust" approaches of building software**

Related To: Availability & Reliability

Some people recommend making your software robust by working around problems automatically. This results in the software “failing slowly.” The program continues working right after an error but fails in strange ways later on.

A system that **fails fast** does exactly the opposite: when a problem occurs, it fails immediately and visibly. Failing fast is a nonintuitive technique: “failing immediately and visibly” sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production.

In overall, the quicker and easier the failure is, the faster it will be fixed. And the fix will be simpler and also more visible. **Fail Fast** is a much better approach for maintainability.

**67: What will you choose: Repository Pattern or "smart" business objects?**

Related To: Design Patterns

Some folks use the "repository pattern" which uses a repository that knows how to fetch, insert, update and delete objects. Those objects are rather "dumb" in that they don't necessarily contain a whole lot of logic - e.g. they're more or less data-transfer objects.

The other camp uses what I call "smart" business objects that know how to load themselves, and they typically have a Save(), possibly Update() or even Delete() method. Here you really don't need any repository - the objects themselves know how to load and save themselves.

What are your thoughts?

**Answer**

* I use the Repository Pattern because of:
* the Single Responsibility Principle. I don't want each individual object having to know how to save, update, delete itself when this can be handled by one single generic repository,
* It also makes unit testing simpler as well,
* data transfer objects (DTO) are more flexible. You can use them everywhere, with no dependency on frameworks, layers, etc.
* the repository pattern doesn't necessarily lead to dumb objects. If the objects have no logic outside Save/Update, you're probably doing too much outside the object. Ideally, you should never use properties to get data from your object, compute things, and put data back in the object. This is a break of encapsulation.

**68: Is Repository Pattern as same as Active Record Pattern?**

Related To: Design Patterns

Big difference between **Active Record and Repository patterns** is in my opinion the owner of the link between entity instance and underlying storage:

Active Record Pattern defines An object that wraps a row in a database table or view, encapsulates the data access, and adds domain logic to that data. In Active Record, entity instance knows how and where to persist itself (this is what "active" means in my mind). That's why you can just call user.save() and it persists itself.

**In the Repository pattern** all of the data access is put in a separate class and is accessed via instance methods. To me, just doing this is beneficial, since data access is now encapsulated in a separate class, leaving the business object to get on with business. This should stop the unfortunate mixing of data access and business logic you tend to get with Active Record. In Repository pattern, entity is more or less dumb POJO, it's the repository that manages its lifecycle. If you create a new instance of the entity, it's not magically persisted, you need to tell the repository to persist it.

**69: How should I be grouping my Repositories when using Repository Pattern?**

One common mistake when using Repository Pattern is to think that table relates to repository 1:1.

Instead, repository should be per Aggregate Root and not a table. It means - if an entity shouldn't live alone (i.e. - if you have a **Registrant** that participates in a particular **Registration**) - it's just an entity and it should be updated/created/retrieved through a repository of Aggregate Root it belongs.

In many cases, this technique of reducing the count of repositories. To avoid that simplification you can cascade repositories through IoC like in this example:

| var registrationService = new RegistrationService(new RegistrationRepository(),  new LicenseRepository(), new GodOnlyKnowsWhatElseThatServiceNeeds()); |
| --- |

**70: What is relationship between Repository and Unit of Work?**

Related To: Design Patterns, Entity Framework

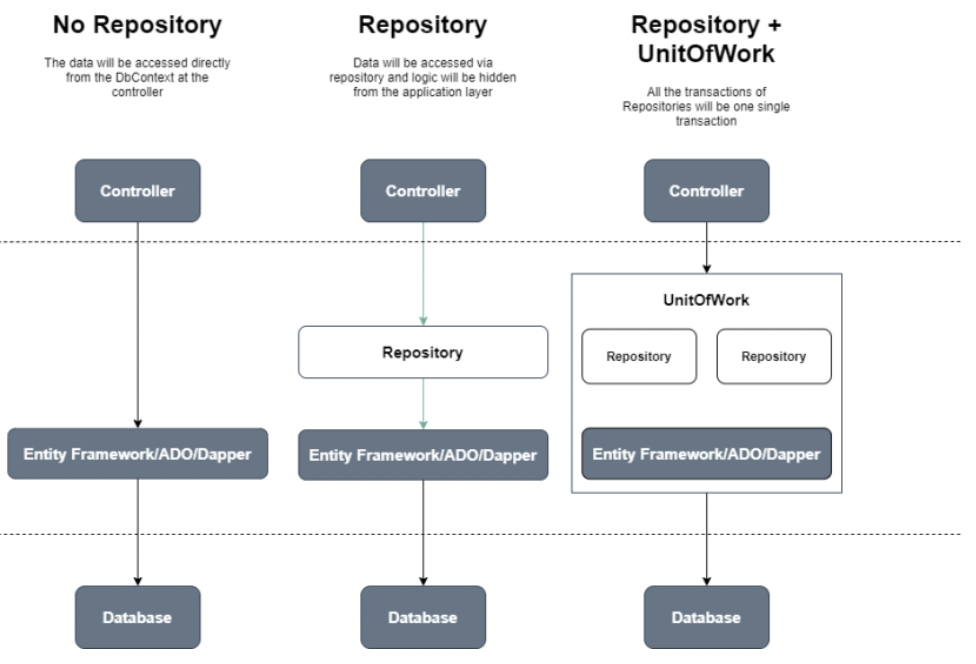
* The Unit of Work pattern "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." UoW orchestrates atomic data operations that span more than one repository. Think of a unit of work as the director (uow) of an orchestra (repositories).

Imagine you had a button called "Delete old records". The button would (roughly) do this:

| using (var uow = GetUnitOfWork())  {  var repository = uow.GetRepository<MyRecord>();  var oldRecords = repository.Entities  .Where(x => x.Old)  .ToList();  foreach(var record in oldRecords)  {  repository.Delete(record);  }  uow.Commit(); // sometimes this is optional (default behavior could be to commit)  } |
| --- |

The responsibilities of the Unit of Work are to:

* Manage transactions.
* Order the database inserts, deletes, and updates.
* Prevent duplicate updates. Inside a single usage of a Unit of Work object, different parts of the code may mark the same Invoice object as changed, but the Unit of Work class will only issue a single UPDATE command to the database.



P.S. The Entity Framework DbContext is a unit of work, and its DbSet<T> properties are repositories.

**71: What is the Dependency Inversion Principle (DIP) and why is it important?**

Related To: Dependency Injection

DIP reduces coupling between different pieces of code and says:

* High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
* Abstractions should never depend upon details. Details should depend upon abstractions.

But why do we invert dependency?

* Basically, we ensure the most stable things are not dependent on less stable things, that might change more frequently.

**72: What does Amdahl's Law mean?** Related To: Reactive Systems

**Amdahl’s law** is a formula used to find the maximum improvement possible by improving a particular part of a system. In parallel computing, Amdahl's law is mainly used to predict the theoretical maximum speedup for program processing using multiple processors. It is named after Gene Amdahl, a computer architect from IBM and the Amdahl Corporation.

**73: What is the most accepted transaction strategy for microservices?**

Related To: Microservices

Microservices introduce eventual consistency issues because of their laudable insistence on decentralized data management. With a monolith, you can update a bunch of things together in a single transaction. Microservices require multiple resources to update, and distributed transactions are frowned upon (for good reason). So now, developers need to be aware of consistency issues, and figure out how to detect when things are out of sync before doing anything the code will regret.

Think how transactions occur and what kind make sense for your services then, you can implement a rollback mechanism that un-does the original operation, or a 2-phase commit system that reserves the original operation until told to commit for real.

Financial services do this kind of thing all the time - if I want to move money from my bank to your bank, there is no single transaction like you'd have in a DB. You don't know what systems either bank is running, so must effectively treat each like your microservices. In this case, my bank would move my money from my account to a holding account and then tell your bank they have some money, if that send fails, my bank will refund my account with the money they tried to send.

**74: Why is writing software difficult? What makes maintaining software hard?**

This is open-ended question. Reference to your experience to provide a relevant answer.

**75: What is the difference between Cohesion and Coupling?** Related To: Microservices

**Cohesion** refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

As for **coupling**, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.

Good software design has **high cohesion and low coupling**.

**76: Are you familiar with The Twelve-Factor App principles?**

**The Twelve-Factor App methodology** is a methodology for building software as a service applications. These best practices are designed to enable applications to be built with portability and resilience when deployed to the web.

1. **Codebase** - There should be exactly one codebase for a deployed service with the codebase being used for many deployments.
2. **Dependencies** - All dependencies should be declared, with no implicit reliance on system tools or libraries.
3. **Config** - Configuration that varies between deployments should be stored in the environment.
4. **Backing services** All backing services are treated as attached resources and attached and detached by the execution environment.
5. **Build, release, run** - The delivery pipeline should strictly consist of build, release, run.
6. **Processes** - Applications should be deployed as one or more stateless processes with persisted data stored on a backing service.
7. **Port binding** - Self-contained services should make themselves available to other services by specified ports.
8. **Concurrency** - Concurrency is advocated by scaling individual processes.
9. **Disposability** - Fast startup and shutdown are advocated for a more robust and resilient system.
10. **Dev/Prod parit**y - All environments should be as similar as possible.
11. **Logs** - Applications should produce logs as event streams and leave the execution environment to aggregate.
12. **Admin Processes** - Any needed admin tasks should be kept in source control and packaged with the application.

**77: What are heuristic exceptions?**

A Heuristic Exception refers to a transaction participant’s decision to unilaterally take some action without the consensus of the transaction manager, usually as a result of some kind of catastrophic failure between the participant and the transaction manager.

In a distributed environment communications failures can happen. If communication between the transaction manager and a recoverable resource is not possible for an extended period of time, the recoverable resource may decide to unilaterally commit or rollback changes done in the context of a transaction. Such a decision is called a heuristic decision. It is one of the worst errors that may happen in a transaction system, as it can lead to parts of the transaction being committed while other parts are rolled back, thus violating the atomicity property of transaction and possibly leading to data integrity corruption.

Because of the dangers of heuristic exceptions, a recoverable resource that makes a heuristic decision is required to maintain all information about the decision in stable storage until the transaction manager tells it to forget about the heuristic decision. The actual data about the heuristic decision that is saved in stable storage depends on the type of recoverable resource and is not standardized. The idea is that a system manager can look at the data, and possibly edit the resource to correct any data integrity problems.

**78: What Does Eventually Consistent Mean?** Related To: Databases

Unlike relational database property of Strict consistency, eventual consistency property of a system ensures that any transaction will eventually (not immediately) bring the database from one valid state to another. This means there can be intermediate states that are not consistent between multiple nodes.

Eventually consistent systems are useful at scenarios where absolute consistency is not critical. For example in case of Twitter status update, if some users of the system do not see the latest status from a particular user its may not be very devastating for system.

Eventually consistent systems can not be used for use cases where absolute/strict consistency is required. For example a banking transactions system can not be using eventual consistency since it must consistently have the state of a transaction at any point of time. Your account balance should not show different amount if accessed from different ATM machines.

**79: What Is Shared Nothing Architecture? How Does It Scale?**

**A shared nothing architecture (SN)** is a distributed computing approach in which each node is independent and self-sufficient, and there is no single point of contention required across the system.

* This means no resources are shared between nodes (No shared memory, No shared file storage)
* The nodes are able to work independently without depending on each other for any work.
* Failure on one node affects only the users of that node, however other nodes continue to work without any disruption.

This approach is highly scalable since it avoid the existence of single bottleneck in the system. Shared nothing is recently become popular for web development due to its linear scalability. Google has been using it for long time.

In theory, A shared nothing system can scale almost infinitely simply by adding nodes in the form of inexpensive machines.

**80: How do I test a private function or a class that has private methods, fields or inner classes?** Related To: Unit Testing

The best way to test a private method is via another public method. If this cannot be done, then one of the following conditions is true:

1. The private method is dead code
2. There is a design smell near the class that you are testing
3. The method that you are trying to test should not be private

Also, by testing private methods you are testing the implementation. This defeats the purpose of unit testing, which is to test the inputs/outputs of a class' contract. A test should only know enough about the implementation to mock the methods it calls on its dependencies. Nothing more. If you can not change your implementation without having to change a test - chances are that your test strategy is poor.

**81: What are best practices for caching paginated results whose ordering/properties can change?** Related To: Caching

It seems what you need is a wrapper for all the parameters that define a page (say, pageNumber, pageSize, sortType, totalCount, etc.) and use this DataRequest object as the key for your caching mechanism. From this point you have a number of options to handle the cache invalidation:

* Implement some sort of timeout mechanism (TTL) to refresh the cache (based on how often the data changes).
* Have a listener that checks database changes and updates the cache based the above parameters (data refresh by server intent).
* If the changes are done by the same process, you can always mark the cache as outdated with every change and check this flag when a page is requested (data refresh by client intent).

The first two might involve a scheduler mechanism to trigger on some interval or based on an event. The last one might be the simpler if you have a single data access point. Lastly, it can quickly become an overly complicated algorithm that outweighs the benefits, so be sure the gain in performance justify the complexity of the algorithm.

**82: Cache miss-storm: Dealing with concurrency when caching invalidates for high-traffic sites**  Related To: Caching

**Problem**

For a high traffic website, there is a method (say **getItems()**) that gets called frequently. To prevent going to the DB each time, the result is cached. However, thousands of users may be trying to access the cache at the same time, and so locking the resource would not be a good idea, because if the cache has expired, the call is made to the DB, and all the users would have to wait for the DB to respond. What would be a good strategy to deal with this situation so that users don't have to wait?

**Answer**

The problem is the so-called **Cache miss-storm (Cache Stampede or Dogpile)** - a scenario in which a lot of users trigger regeneration of the cache, hitting in this way the DB.

To prevent this, first you have to set soft and hard expiration date. Lets say the hard expiration date is 1 day, and the soft 1 hour. The hard is one actually set in the cache server, the soft is in the cache value itself (or in another key in the cache server). The application reads from cache, sees that the soft time has expired, set the soft time 1 hour ahead and hits the database. In this way the next request will see the already updated time and won't trigger the cache update - it will possibly read stale data, but the data itself will be in the process of regeneration.

Next point is: you should have procedure for cache warm-up, e.g. instead of user triggering cache update, a process in your application to pre-populate the new data.

The worst case scenario is e.g. restarting the cache server, when you don't have any data. In this case you should fill cache as fast as possible and there's where a warm-up procedure may play vital role. Even if you don't have a value in the cache, it would be a good strategy to "lock" the cache (mark it as being updated), allow only one query to the database, and handle in the application by requesting the resource again after a given timeout.

**83: Where DTO should be implemented, in a Domain Layer or in an Application Service Layer?** Explain. Related To: DDD, Layering & Middleware

DTOs that are exposed to the outside world become part of a contract. Depending on their form, a good place for them is either the Application Layer or the Presentation Layer.

* If the DTOs are only for presentation purposes, then the Presentation Layer is a good choice.
* If they are part of an API, be it for input or output, that is an Application Layer concern. The Application Layer is what connects your domain model to the outside world.

Don't put your DTO in the Domain Layer. The Domain Layer does not care about mapping things to serve external layers (the domain does not know there is a world outside of its own). The Application Layer is what connects your domain model to the outside world. Presentation Layer should access the domain model only through the Application Layer.

**84: Can we use the CQRS without the Event Sourcing?**

Related To: DDD, Design Patterns

**Event Sourcing (ES)** is optional and in most cases complicates things more than it helps if introduced too early. Especially when transitioning from legacy architecture and even more when the team has no experience with **CQRS**.

Most of the advantages being attributed to **ES** can be obtained by storing your events in a simple Event Log. You don't have to drop your state-based persistence, (but in the long run you probably will, because at some point it will become the logical next step).

My recommendation: Simplicity is the key. Do one step at a time, especially when introducing such a dramatic paradigm shift. Start with simple CQRS, then introduce an Event Log when you (and your team) have become used to the new concepts. Then, if at all required, change your persistence to Event Sourcing.

**85 Could you provide an example of the Single Responsibility Principle?**

Single Responsibility Principle (SRP) states that a class or a method should only be doing one thing and shouldn't be any doing anything related. A class should have only one reason to change.

A typical example could a EmailSender class:

* this should just deal with sending an email out.
* this should not be responsible for loading the email content from database or even formatting the email content to be sent.

Microservices

**1 Define Microservice Architecture**

Microservices, aka Microservice Architecture, is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

**2 List down the advantages of Microservices Architecture**

1. **Independent Development**. All microservices can be easily developed based on their individual functionality
2. **Independent Deployment.** Based on their services, they can be individually deployed in any application
3. **Fault Isolation**. Even if one service of the application does not work, the system still continues to function
4. **Mixed Technology Stack**. Different languages and technologies can be used to build different services of the same application
5. **Granular Scaling**. Individual components can scale as per need, there is no need to scale all components together

**3 Why Would You Opt For Microservices Architecture?**

There are plenty of pros that are offered by Microservices architecture. Here are a few of them:

* Microservices can adapt easily to other frameworks or technologies.
* Failure of a single process does not affect the entire system.
* Provides support to big enterprises as well as small teams.
* Can be deployed independently and in relatively less time.

**4: What are main differences between Microservices and Monolithic Architecture?**

**Microservices**

* Service Startup is fast
* Microservices are loosely coupled architecture.
* Changes done in a single data model does not affect other Microservices.
* Microservices focuses on products, not projects

**Monolithic Architecture**

* Service startup takes time
* Monolithic architecture is mostly tightly coupled.
* Any changes in the data model affect the entire database
* Monolithic put emphasize over the whole project

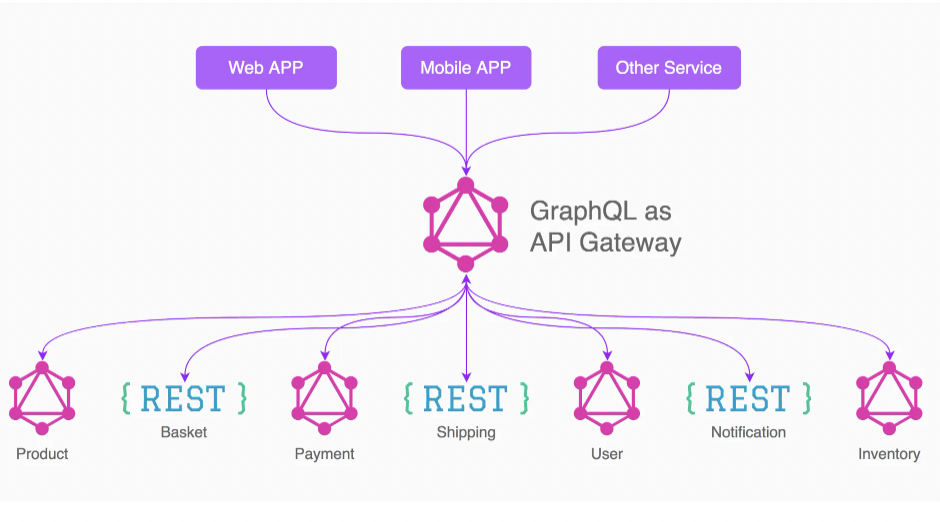
**5: What are the standard patterns of orchestrating microservices?**

As we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services.

* With **orchestration**, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra. The orchestration style corresponds more to the SOA idea of orchestration/task services. For example we could wrap the business flow in its own service. Where the proxy orchestrates the interaction between the microservices like shown in the below picture.
* With **choreography**, we inform each part of the system of its job, and let it work out the details, like dancers all find‐ ing their way and reacting to others around them in a ballet. The choreography style corresponds to the dumb pipes and smart endpoints mentioned by Martin Fowler's. That approach is also called the domain approach and is using domain events, where each service publish events regarding what have happened and other services can subscribe to those events.

**6: Whether do you find GraphQL the right fit for designing microservice architecture?**

GraphQL and microservices are a perfect fit, because GraphQL hides the fact that you have a microservice architecture from the clients. From a backend perspective, you want to split everything into microservices, but from a frontend perspective, you would like all your data to come from a single API. Using GraphQL is the best way I know of that lets you do both. It lets you split up your backend into microservices, while still providing a single API to all your application, and allowing joins across data from different services.



**7: What are smart endpoints and dumb pipes?**

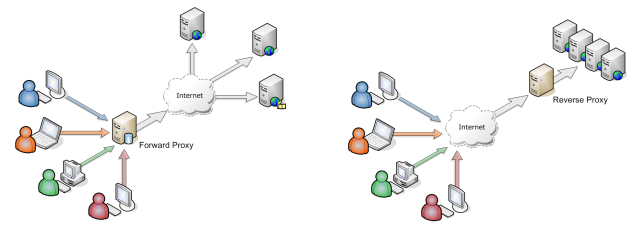
**Smart endpoints** just meaning actual business rules and any other validations happens behind those endpoints which are not visible to anyone to the consumers of those endpoints think of it as a place where actual Magic happens.

**Dumb pipelines** means any communication means where no further actions e.g validations are taken place, it simply carries the data across that particular channel and it may also be replaceable if need be. The infrastructure chosen is typically dumb (dumb as in acts as a message router only). It just means that routing is the only function the pipes should be doing.

**8: What is the difference between a proxy server and a reverse proxy server?**

A simple definition would be:

* **Forward Proxy:** Acting on behalf of a requestor (or service consumer)
* **Reverse Proxy:** Acting on behalf of service/content producer.



**Example**:

Setting up a proxy in your browser so that Netflix doesn't know what country you're in is a forward proxy; an upstream service that directs an incoming request (perhaps you want to send one request to two servers) is a reverse proxy.

**9: What is the difference between Monolithic, SOA and Microservices Architecture?**

* **Monolithic Architecture** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
* **A Service-Oriented Architecture** is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
* **Microservice Architecture** is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

**10: What are the challenges you face while working Microservice Architectures?**

Developing a number of smaller microservices sounds easy, but the challenges often faced while developing them are as follows.

* **Automate the Components**: Difficult to automate because there are a number of smaller components. So for each component, we have to follow the stages of Build, Deploy and, Monitor.
* **Perceptibility**: Maintaining a large number of components together becomes difficult to deploy, maintain, monitor and identify problems. It requires great perceptibility around all the components.
* **Configuration Management**: Maintaining the configurations for the components across the various environments becomes tough sometimes.
* Debugging: Difficult to find out each and every service for an error. It is essential to maintain centralized logging and dashboards to debug problems.

**11: What are the features of Microservices?**

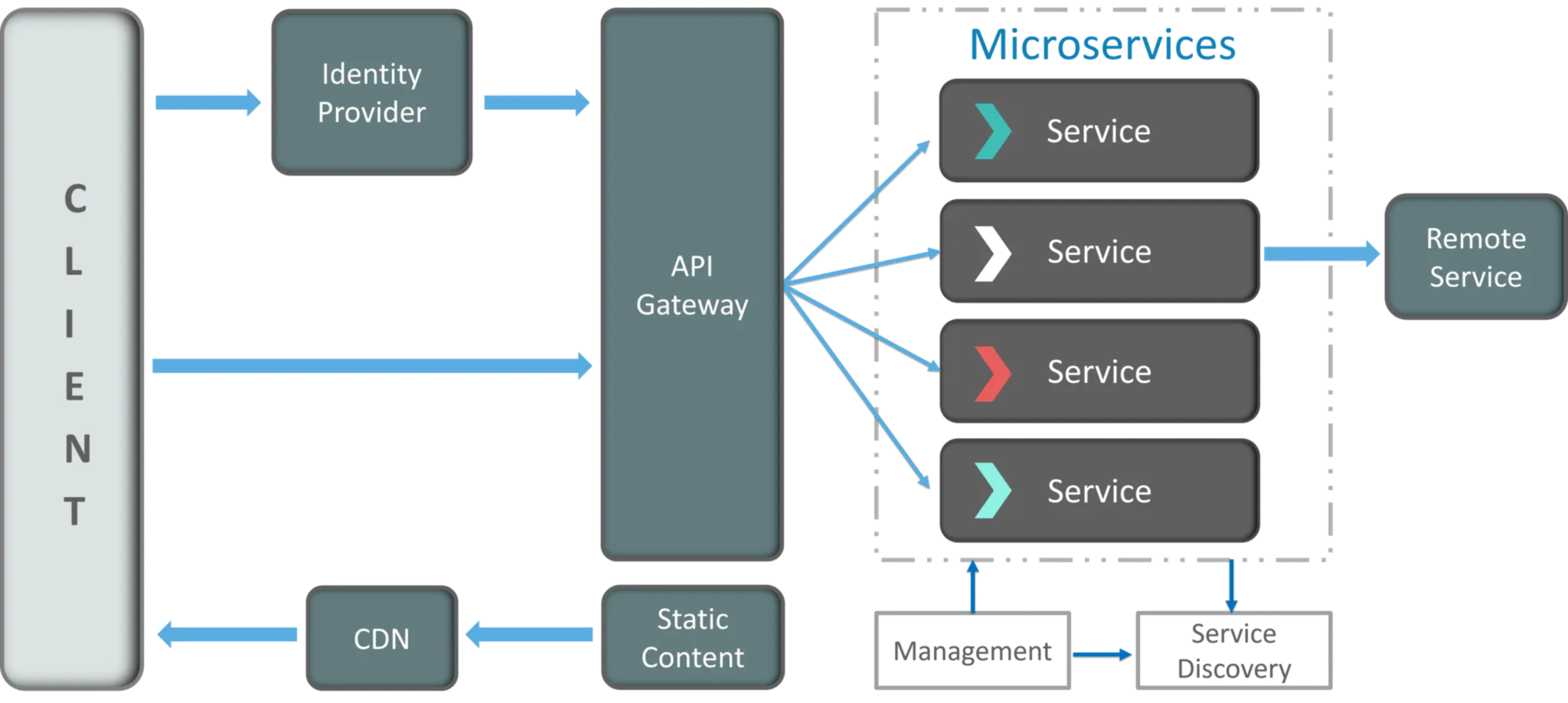
1. **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
2. **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
3. **Business Capabilities** – Microservices are very simple and focus on a single capability
4. **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
5. **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
6. **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
7. **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
8. **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

**12: What Are The Fundamentals Of Microservices Design?**

1. Define a scope
2. Combine loose coupling with high cohesion
3. Create a unique service which will act as an identifying source, much like a unique key in a database table
4. Creating the correct API and take special care during integration.
5. Restrict access to data and limit it to the required level
6. Maintain a smooth flow between requests and response
7. Automate most processes to reduce time complexity
8. Keep the number of tables to a minimum level to reduce space complexity
9. Monitor the architecture constantly and fix any flaw when detected.
10. Data stores should be separated for each microservices.
11. For each microservices, there should be an isolated build.
12. Deploy microservices into containers.
13. Servers should be treated as stateless.

**13: How does Microservice Architecture work?**

* **Clients** – Different users from various devices send requests
* **Identity Providers** – Authenticates user or clients identities and issues security tokens.
* **API Gateway** – Handles client requests.
* **Static Content** – Houses all the content of the system.
* **Management** – Balances services on nodes and identifies failures.
* **Service Discovery** – A guide to find the route of communication between microservices.
* **Content Delivery Networks** – Distributed network of proxy servers and their data centers.
* **Remote Service** – Enables the remote access information that resides on a network of IT devices.



**14: How can we perform Cross-Functional testing?**

Cross-functional testing is verification of non-functional requirements. These requirements are such characteristics of a system that cannot be implemented like a normal feature. Eg. Number of concurrent users supported by system, usability of site etc.

**15: What do you understand by Contract Testing?**

According to Martin Flower, contract test is a test at the boundary of an external service which verifies that it meets the contract expected by a consuming service.

Also, contract testing does not test the behavior of the service in depth. Rather, it tests that the inputs & outputs of service calls contain required attributes and the response latency, throughput is within allowed limits.

**16: How should the various services share a common DB Schema and code?**

The "purest" approach, i.e. the one that gives you the least amount of coupling, is to not share any code. In practice, as usual, it's a tradeoff:

* If the shared functionality is substantial, I'd go for a seperate service.
* If it's just constants, a shared library might be the best solution. You need to be very careful about backwards compatibility, though. Use a packaging system or some source code linkage such as git-tree for distribution.
* For configuration data, you could also implement a specific service, possibly using some existing technology such as LDAP.
* Finally, for simple code that is likely to evolve independently, just duplicating might be the best solution.

Regarding schema - if you want to play by the book, then each microservice has its own database. You don't touch mine, I don't touch yours. That's the better way around this.

**17: Can we create State Machines out of Microservices?**

As we know that each Microservice owning its own database is an independently deployable program unit, this, in turn, lets us create a State Machine out of it. So, we can specify different states and events for a particular microservice.

For Example, we can define an Order microservice. An Order can have different states. The transitions of Order states can be independent events in the Order microservice.

**18: What are the pros and cons of Microservice Architecture?**

**Pros**

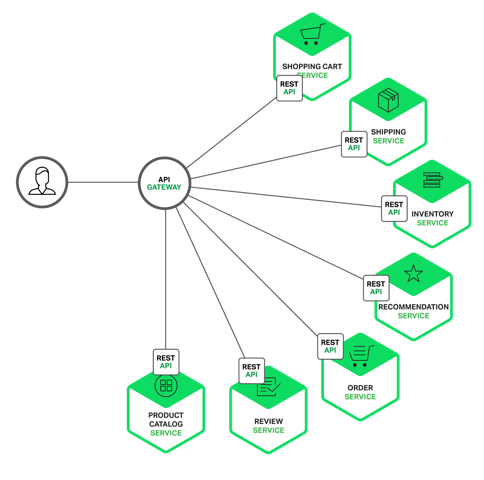
* Freedom to use different technologies
* Each microservices focuses on single capability
* Supports individual deployable units
* Allow frequent software releases
* Ensures security of each service
* Mulitple services are parallelly developed and deployed

**Cons**:

* Increases troubleshooting challenges
* Increases delay due to remote calls
* Increased efforts for configuration and other operations
* Difficult to maintain transaction safety
* Tough to track data across various boundaries
* Difficult to code between services

**19: Explain what is the API Gateway pattern**

An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object‑oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.



**20: What do you understand by Distributed Transaction?**

Distributed Transaction is any situation where a single event results in the mutation of two or more separate sources of data which cannot be committed atomically. In the world of microservices, it becomes even more complex as each service is a unit of work and most of the time multiple services have to work together to make a business successful.

**21: What is the role of an architect in Microservices architecture?**

An architect in microservices architecture plays the following roles:

* Decides broad strokes about the layout of the overall software system.
* Helps in deciding the zoning of the components. So, they make sure components are mutually cohesive, but not tightly coupled.
* Code with developers and learn the challenges faced in day-to-day life.
* Make recommendations for certain tools and technologies to the team developing microservices.
* Provide technical governance so that the teams in their technical development follow principles of Microservice.

**22: Mention some benefits and drawbacks of an API Gateway**

A major benefit of using an API Gateway is that it encapsulates the internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway.

It is yet another highly available component that must be developed, deployed, and

managed. There is also a risk that the API Gateway becomes a development bottleneck.

Developers must update the API Gateway in order to expose each microservice’s endpoints.

**23: What is Materialized View pattern and when will you use it?**

Materialized View pattern is the solution for aggregating data from multiple microservices and used when we need to implement queries that retrieve data from several microservices. In this approach, we generate, in advance (prepare denormalized data before the actual queries happen), a read-only table with the data that's owned by multiple microservices. The table has a format suited to the client app's needs or API Gateway.

A key point is that a materialized view and the data it contains is completely disposable because it can be entirely rebuilt from the source data stores.

This approach not only solves the problem of how to query and join across microservices, but it also improves performance considerably when compared with a complex join, because you already have the data that the application needs in the query table.

**24: What is Idempotence?**

Idempotence refers to a scenario where you perform a task repetitively but the end result remains constant or similar.

**25: What does it mean that shifting to microservices creates a run-time problem?**

**If you are unable to design a monolith that is cleanly divided into components, you will also be unable to design a microservice system.**

Dividing your system into encapsulated, cohesive, and decoupled components is a good idea. It allows you to tackle different problems separately. But you can do that perfectly well in a monolithic deployment (see Fowler: Microservice Premium). After all, this is what OOP has been teaching for many decades! If you decide to turn your components into microservices, you do not gain any architectural advantage. You gain some flexibility regarding technology choice and possibly (but not necessarily!) some scalability. But you are guaranteed some headache stemming from (a) the distributed nature of the system, and (b) the communication between components. Choosing microservices means that you have other problems that are so pressing that you are willing to use microservices despite these problems.

**26: What is a Consumer-Driven Contract (CDC)?**

This is basically a pattern for developing Microservices so that they can be used by external systems. When we work on microservices, there is a particular provider who builds it and there are one or more consumers who use Microservice.

Generally, providers specify the interfaces in an XML document. But in Consumer Driven Contract, each consumer of service conveys the interface expected from the Provider.

**27: What is the most accepted transaction strategy for microservices?**

Microservices introduce eventual consistency issues because of their laudable insistence on decentralized data management. With a monolith, you can update a bunch of things together in a single transaction. Microservices require multiple resources to update, and distributed transactions are frowned upon (for good reason). So now, developers need to be aware of consistency issues, and figure out how to detect when things are out of sync before doing anything the code will regret.

Think how transactions occur and what kind make sense for your services then, you can implement a rollback mechanism that un-does the original operation, or a 2-phase commit system that reserves the original operation until told to commit for real.

Financial services do this kind of thing all the time - if I want to move money from my bank to your bank, there is no single transaction like you'd have in a DB. You don't know what systems either bank is running, so must effectively treat each like your microservices. In this case, my bank would move my money from my account to a holding account and then tell your bank they have some money, if that send fails, my bank will refund my account with the money they tried to send.

**28: What is the difference between Cohesion and Coupling?**

**Cohesion** refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

As for **coupling**, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.

Good software design has **high cohesion** and **low coupling**.

**29: What are Reactive Extensions in Microservices?**

Reactive Extensions also are known as Rx. It is a design approach in which we collect results by calling multiple services and then compile a combined response. These calls can be synchronous or asynchronous, blocking or non-blocking. Rx is a very popular tool in distributed systems which works opposite to legacy flows.

**30: Why would one use sagas over 2PC and vice versa?**

Here are two approaches which I know are used to implement distributed transactions:

* 2-phase commit (2PC)
* Sagas

2PC is a protocol for applications to transparently utilize global ACID transactions by the support of the platform. Being embedded in the platform, it is transparent to the business logic and the application code as far as I know.

Sagas, on the other hand, are series of local transactions, where each local transaction mutates and persist the entities along with some flag indicating the phase of the global transaction and commits the change. In the other words, state of the transaction is part of the domain model. Rollback is the matter of committing a series of "inverted" transactions. Events emitted by the services triggers these local transactions in either case.

* Typically, 2PC is for immediate transactions.
* Typically, Sagas are for long running transactions.

I personally consider Saga capable of doing what 2PC can do, but they have the overhead of implementing the redo mechanism. Opposite is not accurate. I think Sagas are universal, while 2PC involves platform/vendor lockdown and lacks platform independence.

**31: Provide an example of "smart pipes" and "dumb endpoint"**

Components in a system use "pipes" (HTTP/S, queues, etc...) to communicate with each other. Usually these pipes flow through an ESB (Enterprise Service Bus) which does a number of things to the messages being passed between components.

It might do:

* Security checks
* Routing
* Business flow / validation
* Transformation

Once it's completed these tasks the message will be forwarded onto the "endpoint" component. This is an example of "smart pipes" as lots of logic and processing reside inside the ESB (part of the system of "pipes"). The endpoints can then be "dumb" as the ESB has done all the work.

"Smart endpoints and dumb pipes" advocates the opposite scenario. That the lanes of communication should be stripped of business processing and logic and should literally only distribute messages between components. It's then the components themselves that do processing / logic / validation etc... on those messages.

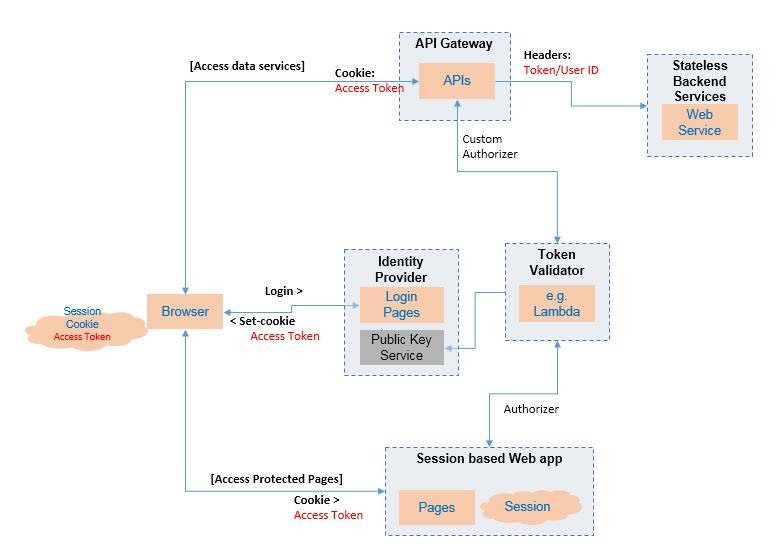
**32: Name the main differences between SOA and Microservices?**

* SOA uses Enterprise Service Bus for communication whereas microservices use much simpler messaging systems.
* Each microservice stores data independently while in SOA components share the same storage.
* For microservices, it’s typical to use Cloud while for SOA Application Servers are much more common.
* SOA is still a monolith, in order to make changes, you need to change the entire architecture.
* SOA using only heavy-weight technologies and protocols (like SOAP, etc) whereas microservices is the leaner, meaner, more agile approach (REST/GraphQL).

**33: What Did The Law Stated By Melvin Conway Implied?**

Conway’s Law applies to modular software systems and states that:

"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization’s communication structure".



**34: How would you implement SSO for Microservice Architecture?**

Add identity service and authorize service access through it using tokens. Any service that has protected resources will talk to the Identity service to make sure the credentials (token) it has are valid. If they are not it will redirect the user for authentication. Once the token had been validated then it could be saved in the session so subsequent calls in the user's session did not have to make the additional call. You can also create a scheduled job if tokens need to be refreshed in that session.

A good way to resolve it is by using the OAuth 2 protocol. In this situation you could authenticate with an OAuth 2.0 endpoint and the token will be added to the HTTP header for calls to your domain. All of the services shall be routed from that domain so you could get the token from the HTTP header.

API Design

**1: What REST stands for?**

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Now a days JSON is the most popular format being used in web services.

**2: What is API Design?**

API design is the process of building an intermediary interface for a system to system connection to expose data to application users and developers.

The fundamental API design influences how well users can consume it and the general user experience. This development process does not allow a single approach. Instead, it combines a series of guidelines to meet initial expectations and continue to work consistently. Application programming interface designers closely follow industry best practices, design patterns, API design principles, and user needs to develop software that presents excellent functionality.

**3: What are the core components of a HTTP Request?**

A HTTP Request has five major parts −

* Verb − Indicate HTTP methods such as GET, POST, DELETE, PUT etc.
* URI − Uniform Resource Identifier (URI) to identify the resource on server.
* HTTP Version − Indicate HTTP version, for example HTTP v1.1 .
* Request Header − Contains metadata for the HTTP Request message as key-value pairs. For example, client ( or browser) type, format supported by client, format of message body, cache settings etc.
* Request Body − Message content or Resource representation.

**4: Define what is SOA**

A Service Oriented Architecture (SOA) is basically defined as an architectural pattern consisting of services. Here application components provide services to the other components using communication protocol over the network. This communication involves data exchanging or some coordination activity between services.

Some of the key principles on which SOA is based are mentioned below

* The service contract should be standardized containing all the description of the services.
* There is loose coupling defining the less dependency between the web services and the client.
* It should follow Service Abstraction rule, which says the service should not expose the way functionality has been executed to the client application.
* Services should be reusable in order to work with various application types.
* Services should be stateless having the feature of discoverability.
* Services break big problems into little problems and allow diverse subscribers to use the services.

**5: What are advantages of REST web services?**

Some of the advantages of REST web services are:

* Learning curve is easy since it works on HTTP protocol
* Supports multiple technologies for data transfer such as text, xml, json, image etc.
* No contract defined between server and client, so loosely coupled implementation.
* REST is a lightweight protocol
* REST methods can be tested easily over browser.

**6: Mention some key characteristics of REST?**

Some key characteristics of REST includes

* REST is stateless, therefore the SERVER has no state (or session data)
* With a well-applied REST API, the server could be restarted between two calls as every data is passed to the server
* Web service mostly uses POST method to make operations, whereas REST uses GET to access resources

**7: Mention whether you can use GET request instead of PUT to create a resource?**

No, you are not supposed to use POST or GET. GET operations should only have view rights

**8: What is cached response?**

Caching refers to storing server response in client itself so that a client needs not to make server request for same resource again and again. A server response should have information about how a caching is to be done so that a client caches response for a period of time or never caches the server response.

**9: What are different types of Web Services?**

There are two types of web services:

* SOAP Web Services: Runs on SOAP protocol and uses XML technology for sending data.
* Restful Web Services: It’s an architectural style and runs on HTTP/HTTPS protocol almost all the time. REST is a stateless client-server architecture where web services are resources and can be identified by their URIs. Client applications can use HTTP GET/POST methods to invoke Restful web services.

**10: Mention what are resources in a REST architecture?**

Resources are identified by logical URLs; it is the key element of a RESTful design. Unlike, SOAP web services in REST, you view the product data as a resource and this resource should contain all the required information.

**11: What are the advantages of Web Services?**

Some of the advantages of web services are:

* **Interoperability**: Web services are accessible over network and runs on HTTP/SOAP protocol and uses XML/JSON to transport data, hence it can be developed in any programming language. Web service can be written in java programming and client can be PHP and vice versa.
* **Reusability**: One web service can be used by many client applications at the same time.
* **Loose Coupling:** Web services client code is totally independent with server code, so we have achieved loose coupling in our application.
* **Easy to deploy and integrate**, just like web applications.
* **Multiple service versions** can be running at same time.

**12: What is SOAP?**

SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it’s XML based, it’s platform and language independent. So our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.

**13: What's the difference between REST & RESTful?**

* Representational state transfer (REST) is a style of software architecture. As described in a dissertation by Roy Fielding, REST is an "architectural style" that basically exploits the existing technology and protocols of the Web.
* RESTful is typically used to refer to web services implementing such an architecture.

**14: WebSockets vs Rest API for real time data? Which to choose?**

**Problem**

I need to constantly access a server to get real time data of financial instruments. The price is constantly changing so I need to request new prices every 0.5 seconds. Which kind if API would you recommend?

**Answer**

The most efficient operation for what you're describing would be to use a webSocket connection between client and server and have the server send updated price information directly to the client over the webSocket ONLY when the price changes by some meaningful amount or when some minimum amount of time has elapsed and the price has changed.

Here's a comparison of the networking operations involved in sending a price change over an already open webSocket vs. making a REST call.

**webSocket**

1. Server sees that a price has changed and immediately sends a message to each client.
2. Client receives the message about new price.

**Rest/Ajax**

1. Client sets up a polling interval
2. Upon next polling interval trigger, client creates socket connection to server
3. Server receives request to open new socket
4. When connection is made with the server, client sends request for new pricing info to server
5. Server receives request for new pricing info and sends reply with new data (if any).
6. Client receives new pricing data
7. Client closes socket
8. Server receives socket close

As you can see there's a lot more going on in the Rest/Ajax call from a networking point of view because a new connection has to be established for every new call whereas the webSocket uses an already open call. In addition, in the webSocket cases, the server just sends the client new data when new data is available - the client doens't have to regularly request it.

A webSocket can also be faster and easier on your networking infrastructure simply because fewer network operations are involved to simply send a packet over an already open webSocket connection versus creating a new connection for each REST/Ajax call, sending new data, then closing the connection. How much of a difference/improvement this makes in your particular application would be something you'd have to measure to really know.

**15: Mention what is the difference between PUT and POST?**

PUT puts a file or resource at a particular URI and exactly at that URI. If there is already a file or resource at that URI, PUT changes that file or resource. If there is no resource or file there, PUT makes one

POST sends data to a particular URI and expects the resource at that URI to deal with the request. The web server at this point can decide what to do with the data in the context of specified resource

PUT is idempotent meaning, invoking it any number of times will not have an impact on resources.

However, POST is not idempotent, meaning if you invoke POST multiple times it keeps creating more resources

**16: What are the best practices to create a standard URI for a web service?**

Following are important points to be considered while designing a URI:

* **Use Plural Noun** − Use plural noun to define resources. For example, we've used users to identify users as a resource.
* **Avoid using spaces** − Use underscore(\_) or hyphen(-) when using a long resource name, for example, use authorized\_users instead of authorized%20users.
* **Use lowercase letters** − Although URI is case-insensitive, it is good practice to keep url in lower case letters only.
* **Maintain Backward Compatibility** − As Web Service is a public service, a URI once made public should always be available. In case, URI gets updated, redirect the older URI to new URI using HTTP Status code, 300.
* **Use HTTP Verb** − Always use HTTP Verb like GET, PUT, and DELETE to do the operations on the resource. It is not good to use operations names in URI.

**17: What is the use of Accept and Content-Type Headers in HTTP Request?**

* **Accept headers** tells web service what kind of response client is accepting, so if a web service is capable of sending response in XML and JSON format and client sends Accept header as application/xml then XML response will be sent. For Accept header application/json, server will send the JSON response.
* **Content-Type header** is used to tell server what is the format of data being sent in the request. If Content-Type header is application/xml then server will try to parse it as XML data. This header is useful in HTTP Post and Put requests.

**18: What is statelessness in RESTful Webservices?**

As per REST architecture, a RESTful web service should not keep a client state on server. This restriction is called statelessness. It is responsibility of the client to pass its context to server and then server can store this context to process client's further request. For example, session maintained by server is identified by session identifier passed by the client.

**19: What is Payload?**

The request data which is present in the body part of every HTTP message is referred as Payload. In Restful web service, the payload can only be passed to the recipient through POST method.

There is no limit of sending data as payload through POST method but the only concern is that more data with consuming more time and bandwidth. This may consume much of user’s time also.

2**0: Mention what is the difference between RPC or document style web services? How you determine to which one to choose?**

In document style web services, we can transport an XML message as part of SOAP request which is not possible in RPC style web service. Document style web service is most appropriate in some application where XML message behaves as document and content of that document can alter and intention of web service does not rely on the content of XML message.

**21: What are the best practices for caching?**

Always keep static contents like images, css, JavaScript cacheable, with expiration date of 2 to 3 days. Never keep expiry date too high.

Dynamic contents should be cached for few hours only.

**22: What are different ways to test web services?**

* SOAP web services can be tested programmatically by generating client stubs from WSDL or through software such as Soap UI.
* REST web services can be tested easily with program, curl commands and through browser extensions. Resources supporting GET method can be tested with browser itself, without any program.

**23: What are the best practices to design a resource representation?**

Following are important points to be considered while designing a representation format of a resource in a RESTful web services −

* Understandability − Both Server and Client should be able to understand and utilize the representation format of the resource.
* Completeness − Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.
* Linkablity − A resource can have a linkage to another resource, a format should be able to handles such situations.

**24: What is the purpose of HTTP Status Code?**

HTTP Status code are standard codes and refers to predefined status of task done at server. For example, HTTP Status 404 states that requested resource is not present on server.

Consider following status codes:

* **200** - OK, shows success.
* **201** - CREATED, when a resource is successful created using POST or PUT request. Return link to newly created resource using location header.
* **304** - NOT MODIFIED, used to reduce network bandwidth usage in case of conditional GET requests. Response body should be empty. Headers should have date, location etc.
* **400** - BAD REQUEST, states that invalid input is provided e.g. validation error, missing data.
* **401** - FORBIDDEN, states that user is not having access to method being used for example, delete access without admin rights.
* **404** - NOT FOUND, states that method is not available.
* **409** - CONFLICT, states conflict situation while executing the method for example, adding duplicate entry.
* **500** - INTERNAL SERVER ERROR, states that server has thrown some exception while executing the method.

**25: What are disadvantages of SOAP Web Services?**

Some of the disadvantages of SOAP protocol are:

* Only XML can be used, JSON and other lightweight formats are not supported.
* SOAP is based on the contract, so there is a tight coupling between client and server applications.
* SOAP is slow because payload is large for a simple string message, since it uses XML format.
* Anytime there is change in the server side contract, client stub classes need to be generated again. Can’t be tested easily in browser

**26: What is UDDI? Related to SOA**

UDDI is acronym for Universal Description, Discovery and Integration. UDDI is a directory of web services where client applications can lookup for web services. Web Services can register to the UDDI server and make them available to client applications.

**27: What are the disadvantages of statelessness in RESTful Webservices?**

Following is the disadvantage of statelessness in RESTful web services:

Web services need to get extra information in each request and then interpret to get the client's state in case client interactions are to be taken care of.

**28: Mention what are the different application integration styles?**

The different integration styles include

* Shared database
* Batch file transfer
* Invoking remote procedure (RPC)
* Swapping asynchronous messages over a message oriented middle-ware (MOM)

**29: How would you choose between SOAP and REST web services?**

Web Services work on client-server model and when it comes to choose between SOAP and REST, it all depends on project requirements. Let’s look at some of the conditions affecting our choice:

* Do you know your web service clients beforehand? If Yes, then you can define a contract before implementation and SOAP seems better choice. But if you don’t then REST seems better choice because you can provide sample request/response and test cases easily for client applications to use later on.
* How much time you have? For quick implementation REST is the best choice. You can create web service easily, test it through browser/curl and get ready for your clients. What kind of data format are supported? If only XML then you can go with SOAP but if you think about supporting JSON also in future then go with REST.

**30: What are the primary security issues of web service?**

To ensure reliable transactions and secure confidential information, web services requires very high level of security which can be only achieved through Entrust Secure Transaction Platform. Security issues for web services are broadly divided into three sections as described below

1) **Confidentiality**: A single web service can have multiple applications and their service path contains a potential weak link at its nodes. Whenever messages or say XML requests are sent by the client along with the service path to the server, they must be encrypted. Thus, maintaining the confidentiality of the communication is a must.

2) **Authentication**: Authentication is basically performed to verify the identity of the users as well as ensuring that the user using the web service has the right to use or not? Authentication is also done to track user’s activity. There are several options that can be considered for this purpose

* Application level authentication
* HTTP digest and HTTP basic authentication
* Client certificates

3) **Network Security**: This is a serious issue which requires tools to filter web service traffic.

**31: What are the core components of a HTTP response?**

A HTTP Response has four major parts −

* Status/Response Code − Indicate Server status for the requested resource. For example 404 means resource not found and 200 means response is ok.
* HTTP Version − Indicate HTTP version, for example HTTP v1.1 .
* Response Header − Contains metadata for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.
* Response Body − Response message content or Resource representation.

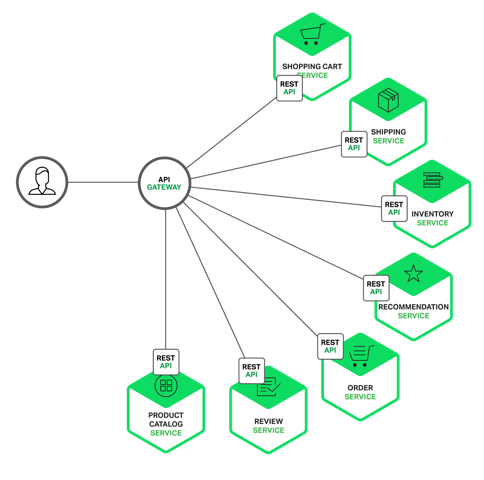
**32: What are disadvantages of REST web services?**

Some of the disadvantages of REST are:

* Since there is no contract defined between service and client, it has to be communicated through other means such as documentation or emails.
* Since it works on HTTP, there can’t be asynchronous calls.
* Sessions can’t be maintained.

**33 Explain what is the API Gateway pattern**

An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object‑oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.



A major benefit of using an API Gateway is that it encapsulates the internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway.

**34: Which header of HTTP response provides control over caching?**

**Cache-Control** is the primary header to control caching.

**35: What do you mean by idempotent operation?**

Idempotent operations means their result will always same no matter how many times these operations are invoked.

**36: Explain element?**

Definition element is described as the root of WSDL document which defines the name of the web service as well as act as a container for all the other elements.

**37: What are the advantages of statelessness in RESTful Webservices?**

Following are the benefits of statelessness in RESTful web services:

* Web services can treat each method request independently.
* Web services need not to maintain client's previous interactions. It simplifies application design.
* As HTTP is itself a statelessness protocol, RESTful Web services work seamlessly with HTTP protocol.

**38: What is difference between SOA and Web Services?**

Service Oriented Architecture (SOA) is an architectural pattern where applications are designed in terms of services that can be accessed through communication protocol over network. SOA is a design pattern and doesn’t go into implementation.

Web Services can be thought of as Services in SOAP architecture and providing means to implement SOA pattern.

**39: Explain Cache-control header**

A standard Cache control header can help in attaining cache ability. Enlisted below is the brief description of various cache control header:

* **Public**: Resources that are marked as the public can be cached by any intermediate components between the client and server.
* **Private**: Resources that are marked as private can only be cached by the client.
* **No cache** means that particular resource cannot be cached and thus the whole process is stopped.

**40: Which type of Webservices methods are to be idempotent?**

PUT and DELETE operations are idempotent

**41: What is difference between OData and REST web services?**

* REST - is an architecture of how to send messages over HTTP.
* OData (v4) - is a specific implementation of REST, really defines the content of the messages in different formats (currently I think is AtomPub and JSON). ODataV4 follows rest principles.

For example, asp.net people will mostly use WebApi controller to serialize/deserialize objects into JSON and have javascript do something with it. The point of Odata is being able to query directly from the URL with out-of-the-box options.

**42: Explain the difference between WCF, Web API, WCF REST and Web Service?**

The .Net framework has a number of technologies that allow you to create HTTP services such as Web Service, WCF and now Web API. There are a lot of articles over the internet which may describe to whom you should use.

**Web Service**

* It is based on SOAP and return data in XML form.
* It support only HTTP protocol.
* It is not open source but can be consumed by any client that understands xml.
* It can be hosted only on IIS.

**WCF**

* It is also based on SOAP and return data in XML form.
* It is the evolution of the web service(ASMX) and support various protocols like TCP, HTTP, HTTPS, Named Pipes, MSMQ.
* The main issue with WCF is, its tedious and extensive configuration.
* It is not open source but can be consumed by any client that understands xml.
* It can be hosted with in the applicaion or on IIS or using window service.

**WCF Rest**

* To use WCF as WCF Rest service you have to enable webHttpBindings.
* It support HTTP GET and POST verbs by WebGet and WebInvoke attributes respectively.
* To enable other HTTP verbs you have to do some configuration in IIS to accept request of that particular verb on .svc files
* Passing data through parameters using a WebGet needs configuration. The UriTemplate must be specified.
* It support XML, JSON and ATOM data format.

**Web API**

* This is the new framework for building HTTP services with easy and simple way.
* Web API is open source an ideal platform for building REST-ful services over the .NET Framework.
* Unlike WCF Rest service, it use the full feature of HTTP (like URIs, request/response headers, caching, versioning, various content formats)
* It also supports the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection, unit testing

**43: What are the best practices to be followed while designing a secure RESTful web service?**

As RESTful web services work with HTTP URLs Paths so it is very important to safeguard a RESTful web service in the same manner as a website is be secured. Following are the best practices to be followed while designing a RESTful web service:

* **Validation** − Validate all inputs on the server. Protect your server against SQL or NoSQL injection attacks.
* **Session based authentication** − Use session based authentication to authenticate a user whenever a request is made to a Web Service method.
* **No sensitive data in URL** − Never use username, password or session token in URL , these values should be passed to Web Service via POST method.
* **Restriction on Method execution** − Allow restricted use of methods like GET, POST, DELETE. GET method should not be able to delete data.
* **Validate Malformed XML/JSON** − Check for well formed input passed to a web service method.
* **Throw generic Error Messages** − A web service method should use HTTP error messages like 403 to show access forbidden etc.

**44: Enlist some important constraints for RESTful web services**

* Every constraint has positive as well as negative impacts and to produce an overall architecture, there should be the balance between both of them. Below mentioned are some important constraints for RESTful web service:
* There should be separate concerns for each server and client which will help to maintain the modularity within the application. This will also reduce the complexity and increase the scalability.
* The client-server communication should be stateless, which means no previous information is used and the complete execution is done in isolation. In cases of failure, it also helps the client to recover.
* In client-server communication, the HTTP response should be cacheable so that when required cached copy can be used which in turn enhances the scalability and performance of the server.
* The fourth constraint is the uniform interface which allows client-server interaction to be easily understood. This constraint is further divided into four sub-constraints as:
  + Resource Identification
  + Resource manipulation
  + Each message is easily understood and is self-descriptive.
  + Hypermedia, which is defined as the text with hyperlinks and when clicked it moves to another application state.
* Client-server communication should be done on a layered system and thus the client should only have knowledge about the intermediate level with which communication is being done.

**45: What is Open API Initiative?**

The Open API Initiative was created by an industry consortium to standardize REST API descriptions across vendors. As part of this initiative, the Swagger 2.0 specification was renamed the OpenAPI Specification (OAS) and brought under the Open API Initiative.

You may want to adopt OpenAPI for your web APIs. Some points to consider:

* The OpenAPI Specification comes with a set of opinionated guidelines on how a REST API should be designed. That has advantages for interoperability, but requires more care when designing your API to conform to the specification.
* OpenAPI promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.
* Tools like Swagger can generate client libraries or documentation from API contracts. For example, see ASP.NET Web API Help Pages using Swagger.

**46: Name some best practices for better RESTful API design Related To: REST & RESTful**

* Use nouns and HTTP methods but no verbs

| GET /cars  POST /cars  DELETE /cars/:id  instead  GET /getAllCars  POST /createNewCar  GET /deleteAllRedCars |
| --- |

* GET method and query parameters should not alter the state
* Use plural noun

| /cars instead of /car  /users instead of /user  /products instead of /product  /settings instead of /setting |
| --- |

* Use sub-resources for relations

| GET /cars/711/drivers/ Returns a list of drivers for car 711  GET /cars/711/drivers/4 Returns driver #4 for car 711 |
| --- |

* Use HTTP headers for serialisation format

| Content-Type defines the request format.  Accept defines a list of acceptable response formats. |
| --- |

* Use HATEOAS - Hypermedia as the Engine of Application State is a principle that hypertext links should be used to create a better navigation through the API.

| {  "id": 711,  "manufacturer": "bmw",  "model": "X5",  "seats": 5,  "drivers": [{  "id": "23",  "name": "Stefan Jauker",  "links": [{  "rel": "self",  "href": "/api/v1/drivers/23"  }]  }]  } |
| --- |

* Use appropriate HTTP response status codes
* 2xx (Success category)
* 3xx (Redirection Category)
* 4xx (Client Error Category)
* 5xx (Server Error Category)
* Provide filtering, sorting, field selection and paging for collections

| GET /cars?color=red Returns a list of red cars  GET /cars?seats<=2 Returns a list of cars with a maximum of 2 seats |
| --- |

* Version your API

| /blog/api/v1 |
| --- |

* Use error payloads - All exceptions should be mapped in an error payload.

| {  "errors": [{  "userMessage": "Sorry, the requested resource does not exist",  "internalMessage": "No car found in the database",  "code": 34,  "more info": "http://dev.mwaysolutions.com/blog/api/v1/errors/12345"  }]  } |
| --- |

* Allow overriding HTTP method

In certain situations (for example, when the service or its consumers are behind an overzealous corporate firewall, or if the main consumer is a web page), only the GET and POST HTTP methods might be available. In such a case, it is possible to emulate the missing verbs by passing a custom header in the requests. To support a RESTful API with these limitations, the API needs a way to override the HTTP method and use the custom HTTP Header X-HTTP-Method-Override to map the request to an appropriate API method.

SOA

**1: Define what is SOA**

A Service Oriented Architecture (SOA) is basically defined as an architectural pattern consisting of services. Here application components provide services to the other components using communication protocol over the network. This communication involves data exchanging or some coordination activity between services.

Some of the key principles on which SOA is based are mentioned below

* The service contract should be standardized containing all the description of the services.
* There is loose coupling defining the less dependency between the web services and the client.
* It should follow Service Abstraction rule, which says the service should not expose the way functionality has been executed to the client application.
* Services should be reusable in order to work with various application types.
* Services should be stateless having the feature of discoverability.
* Services break big problems into little problems and allow diverse subscribers to use the services.

**2: What is WSDL?**

WSDL stands for Web Service Description Language. WSDL is an XML based document that provides technical details about the web service. Some of the useful information in WSDL document are:

* method name,
* port types,
* service end point,
* binding,
* method parameters etc.

**3: What is SOAP?**

SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it’s XML based, it’s platform and language independent. So our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.

**4: What is the difference between Monolithic, SOA and Microservices Architecture?**

* **Monolithic Architecture** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
* **A Service-Oriented Architecture** is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
* **Microservice Architecture** is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

**5: Explain WSDL?**

WSDL stands for Web service Description Language. It is a simple XML document which comes under the Service Description layer of Web Service Protocol Stock and describes the technical details or locates the user interface to web service. Few of the important information present in WSDL document are

* Method name
* Port types
* Service end point
* Method parameters
* Header information
* Origin, etc

**6: What are advantages of SOAP Web Services?**

SOAP web services have all the advantages that web services has, some of the additional advantages are:

* **WSDL document** provides contract and technical details of the web services for client applications without exposing the underlying implementation technologies.
* **SOAP uses XML** data for payload as well as contract, so it can be easily read by any technology.
* **SOAP protocol** is universally accepted, so it's an industry standard approach with many easily available open source implementations.

**7: What are disadvantages of SOAP Web Services?**

Some of the disadvantages of SOAP protocol are:

* Only XML can be used, JSON and other lightweight formats are not supported.
* SOAP is based on the contract, so there is a tight coupling between client and server applications.
* SOAP is slow because payload is large for a simple string message, since it uses XML format.
* Anytime there is change in the server side contract, client stub classes need to be generated again. Can’t be tested easily in browser

**8: What is UDDI?**

UDDI is acronym for Universal Description, Discovery and Integration. UDDI is a directory of web services where client applications can lookup for web services. Web Services can register to the UDDI server and make them available to client applications.

**9: How would you choose between SOAP and REST web services?**

Web Services work on client-server model and when it comes to choose between SOAP and REST, it all depends on project requirements. Let’s look at some of the conditions affecting our choice:

* Do you know your web service clients beforehand? If Yes, then you can define a contract before implementation and SOAP seems better choice. But if you don’t then REST seems better choice because you can provide sample request/response and test cases easily for client applications to use later on.
* How much time you have? For quick implementation REST is the best choice. You can create web service easily, test it through browser/curl and get ready for your clients. What kind of data format are supported? If only XML then you can go with SOAP but if you think about supporting JSON also in future then go with REST.

**10: What are the various approaches available for developing SOAP based web services?**

There are basically 2 different approaches available for developing SOAP-based web services. These are explained as follows

* Contract-first approach: In this approach, the contract is defined first by XML and WSDL and then java classes are derived from the contract.
* Contract-last approach: In this approach, java classes are defined first and then the contract is generated which is usually the WSDL file from the java class.

“Contract-first” method is the most preferred approach.

**11: What are different components of WSDL?**

Some of the different tags in WSDL xml are:

* **xsd:import** namespace and **schemaLocation**: provides WSDL URL and unique namespace for web service.
* **message**: for method arguments
* **part**: for method argument name and type
* **portType**: service name, there can be multiple services in a wsdl document.
* **operation**: contains method name
* **soap**:address for endpoint URL.

**12: What are the two attributes of element in WSDL?**

Every port element is related to a specific binding by defining an individual endpoint. The port element has following two attributes

* **Name**: This attribute provides the unique name within the WSDL document.
* **Binding**: This attribute refers to the process of binding which has to be performed as per the linking rules defined by WSDL.

**13: What are the different elements of WSDL documents?**

The different elements of WSDL document along with brief description is enlisted below

* **Types**: This defines the message data types, which are in the form of XML schema, used by the web services.
* **Message**: This defines the data elements for each operation where messages could be the entire document or an argument that is to be mapped.
* **Port Type**: There are multiple services present in WSDL. Port type defines the collection of operations that can be performed for binding.
* **Binding**: Determines and defines the protocol and data format for each port type.
* **Operations**: This defines the operations performed for a message to process the message.

**14: Enlist the operation types response used in WSDL?**

WSDL basically defines 4 types of Operation type responses. These are enlisted below

* **One-way**: Receives a message but does not return response.
* **Request-Response:** Receives a request and return a response.
* **Solicit-Response:** Sends a request and wait for a response.
* **Notification**: Sends a message but does not wait for a response.

Among these, Request-Response is the most common operation type.

**15: Explain the message element in WSDL?**

Message element describes the data that has been exchanged between the consumer and the web service providers. Every web service consists of two messages and each message has zero or more parameters. The two messages are

* **Input**: Describes the parameter for the web service
* **Output**: Describes the return data from the web service.

**16: What are the elements of a SOAP message?**

SOAP is just like other XML document and has following elements

* **Envelope**: This element is defined as the mandatory root element. It translates the XML document and determines the start and end of the SOAP message.
* **Header**: This element contains the optional header attributes of the message that contains specific information of the application. This element can occur multiple times and are intended to add new features and functionalities.
* **Body**: This element is mandatory and contains the call and response messages. It is also defined as the child element of the envelope containing all the application derived XML data that has been exchanged as a part of SOAP message.
* **Fault element**: Errors that occur during processing of the messages are handled by the fault element. If the error is present, then this element appears as a child element of the body. However, there can only be one fault block.

**17: What is difference between Top Down and Bottom Up approach in SOAP Web Services?**

**In Top Down approach** first WSDL document is created to establish the contract between web service and client and then code is written, it’s also termed as contract first approach. This is hard to implement because classes need to be written to confirm the contract established in WSDL. Benefit of this approach is that both client and server code can be written in parallel.

**In Bottom Up approach,** first web service code is written and then WSDL is generated. It’s also termed as contract last approach. This approach is easy to implement because WSDL is generated based on code. In this approach client code have to wait for WSDL from server side to start their work.

**18: What is difference between SOA and Web Services?**

**Service Oriented Architecture (SOA) i**s an architectural pattern where applications are designed in terms of services that can be accessed through communication protocol over network. SOA is a design pattern and doesn’t go into implementation.

**Web Services c**an be thought of as Services in SOAP architecture and providing means to implement SOA pattern.

**19: What are the important characteristics of SOAP envelope element?**

* SOAP envelope is a packaging mechanism.
* Every SOAP message has a mandatory root envelope message.
* Only one body element is allowed for each envelope element.
* As the SOAP version changes, envelope changes.
* If the header element is present, it should appear as the first child.
* Prefix ENV and envelope element is used for specification.
* A namespace and an optional encoding style are used in case of optional SOAP encoding.

**20: Name the main differences between SOA and Microservices?**

* SOA uses Enterprise Service Bus for communication whereas microservices use much simpler messaging systems.
* Each microservice stores data independently while in SOA components share the same storage.
* For microservices, it’s typical to use Cloud while for SOA Application Servers are much more common.
* SOA is still a monolith, in order to make changes, you need to change the entire architecture.
* SOA using only heavy-weight technologies and protocols (like SOAP, etc) whereas microservices is the leaner, meaner, more agile approach (REST/GraphQL).

**21: Is binding between SOAP and WSDL possible?**

Yes, it is possible to bind WSDL to SOAP. The binding is possible by basically two attributes

* **Name**: Defines the name of the binding.
* **Type**: Defines the port for the binding.

For SOAP binding, two attributes need to be declared

* **Transport**: Defines the SOAP protocol to be used i.e. HTTP.
* **Style**: This attribute can be ‘rpc’ or ‘document’.

Availability & Reliability

**1: What is Availability?**

Availability refers to the probability that a system performs correctly at a specific time instance (not duration). Interruptions may occur before or after the time instance for which the system’s availability is calculated. The service must be operational and adequately satisfy the defined specifications at the time of its usage.

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. Availability is generally measured in number of 9s--a service with 99.99% availability is described as having four 9s.

**2: What is Reliability?**

**Reliability** is the probability that a system performs correctly during a specific time duration. During this correct operation, no repair is required or performed, and the system adequately follows the defined performance specifications.

**Reliability** follows an exponential failure law, which means that it reduces as the time duration considered for reliability calculations elapses. In other words, reliability of a system will be high at its initial state of operation and gradually reduce to its lowest magnitude over time.

**3: What is Back-Pressure?**

When one component is struggling to keep-up, the system as a whole needs to respond in a sensible way. It is unacceptable for the component under stress to fail catastrophically or to drop messages in an uncontrolled fashion. Since it can’t cope and it can’t fail it should communicate the fact that it is under stress to upstream components and so get them to reduce the load.

This **back-pressure is an important feedback mechanism that allows systems to gracefully respond** to load rather than collapse under it. The back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load.

**4: How Do you update a live heavy traffic site with minimum or Zero Down Time?**

Deploying a newer version of a live website can be a challenging task specially when a website has high traffic. Any downtime is going to affect the users. There are a few best practices that we can follow:

**Before deploying on Production:**

* Thoroughly test the new changes and ensure it working in a test environment which is almost identical to production system.
* If possible do automation of test cases as much as possible.
* Create a automated sanity testing script (also called as smoke test) that can be run on production (without affecting real data). These are typically readonly type of test cases. However depending on your application needs you can add more cases to this. Make sure it can be run quickly by keeping it short.
* Create scripts for all manual tasks(if possible), avoiding any hand typing mistakes during day of deployment.
* Test the script to make sure they work on a non-production environment.
* Keep the build artifacts ready. e.g application deployment files, database scripts, config files etc.
* Create a checklist of things to do on day of deployment.
* Rehearse. Deploy in a non-prod environment is almost identical to production. Try this with production data volumes(if possible). Make a note of time required for your tasks so you can plan accordingly.

**When doing deploying on a production environment:**

* Use Green-Blue deployment technique to reduce down-time risk
* Keep backup of current site/data to be able to rollback
* Use sanity test cases before doing a lot of in depth testing

**5: What Do You Mean By High Availability (HA)?**

**Availability** means the ability of the application user to access the system, If a user cannot access the application, it is assumed unavailable. High Availability means the application will be available, without interruption. Using redundant server nodes with clustering is a common way to achieve higher level of availability in web applications.

Availability is commonly expressed as a percentage of uptime in a given year.

**6: What does it mean "System Shall Be Resilient"?**

System is Resilient if it stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure.

Resilience is achieved by:

* replication,
* containment,
* isolation and
* delegation.

Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

**7: What is Fail-over?**

In computing and related technologies such as networking, failover is switching to a redundant or standby computer server, system, hardware component or network upon the failure or abnormal termination of the previously active application,1 server, system, hardware component, or network.

**Failover** is a means of achieving high availability (HA). Think of HA as a feature and failover as one possible implementation of that feature. Failover is not always the only consideration when achieving HA.

There are two complementary patterns to support high availability: fail-over and replication.

**8: Explain Failure in contrast to Error**

* **A failure** is an unexpected event within a service that prevents it from continuing to function normally. A failure will generally prevent responses to the current, and possibly all following, client requests.
* This is in contrast with an **error**, which is an expected and coded-for condition—for example an error discovered during input validation, that will be communicated to the client as part of the normal processing of the message.

Failures are unexpected and will require intervention before the system can resume at the same level of operation. This does not mean that failures are always fatal, rather that some capacity of the system will be reduced following a failure. Errors are an expected part of normal operations, are dealt with immediately and the system will continue to operate at the same capacity following an error.

**9: How to choose between CP (consistency) and AP (availability)?**

* You need consistency (or what is called a CP database) if the data in the database must always be up to date and aligned, even in the instance of a network failure (eg. the partitioned nodes are unable to communicate with one another for whatever reason).
* Particular use cases where you would prioritize consistency is when you need multiple clients to have the same view of the data. For example, where you’re dealing with financial information, personal information, using a database that gives you consistency and confidence that data you are looking at is up to date in a situation where the network is unreliable or fails.
* You need availability when data accumulation is a priority. In scenarios like these, you will want to capture as much information as possible, but it isn’t critical that the database is constantly up to date. It simply just needs to be accessible and available even when network connections aren’t working.

**10: Explain how does Active-Passive Fail-over work?**

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.

The length of downtime is determined by whether the passive server is already running in 'hot' standby or whether it needs to start up from 'cold' standby. Only the active server handles traffic.

Active-passive failover can also be referred to as master-slave failover.

**11: What is Active-Active Fail-over?**

In active-active, both servers are managing traffic, spreading the load between them.

If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.

Active-active failover can also be referred to as master-master failover.

**12: Compare "Fail Fast" vs "Robust" approaches of building software**

Some people recommend making your software robust by working around problems automatically. This results in the software “failing slowly.” The program continues working right after an error but fails in strange ways later on.

A system that fails fast does exactly the opposite: when a problem occurs, it fails immediately and visibly. Failing fast is a nonintuitive technique: “failing immediately and visibly” sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production.

In overall, the quicker and easier the failure is, the faster it will be fixed. And the fix will be simpler and also more visible. Fail Fast is a much better approach for maintainability.

**13: Explain how to calculate Availability of multiple system components**

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

**In sequence**

Overall availability decreases when two components with availability < 100% are in sequence:

| Availability (Total) = Availability (Foo) \* Availability (Bar) |
| --- |

If both Foo and Bar each had 99.9% availability, their total availability in sequence would be 99.8%.

**In parallel**

Overall availability increases when two components with availability < 100% are in parallel:

| Availability (Total) = 1 - (1 - Availability (Foo)) \* (1 - Availability (Bar)) |
| --- |

If both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

**14: What is a crashloop?**

A crashloop is when a process crashes and is restarted by a watchdog daemon, indefinitely.

That is, the history is:

* Process starts at time T.
* Process crashes at time T+1.
* Watchdog daemon restarts process.
* Process started at time T+2.
* Process crashes at time T+3.
* Watchdog daemon restarts process.
* Process starts...etc.

In general, in distributed computing if you want something to eventually succeed, you have to write down your intent for it to be completed and you need a worker to loop continually to act on this intent. This is "at least once delivery" of a work item.

Here, the intent is that the task runs, and watchdog itself is running the loop that is constantly trying to make sure the task runs. This is why when a task crashes, it is restarted. When a task crashes repeatedly, together you end up with a crashloop.

CDN

**1: What is a CDN?**

**A CDN (content delivery network)** is essentially a group of servers that are strategically placed across the globe with the purpose of accelerating the delivery of your static web content.

For website owners who have visitors in multiple geographic locations, content will be delivered faster to these users as there is less distance to travel. CDN users also benefit from the ability to easily scale up and down much more easily due to traffic spikes. On average, 80% of a website consist of static resources therefore when using a CDN, there is much less load on the origin server.

**2: Why use a CDN (Content Delivery Network‎)?**

CDNs are very useful for a multiple reasons:

* **It simply Decrease Server Load** - On average, 80% of a website consist of static resources therefore when using a CDN, there is much less load on the origin server.
* **Make Faster Content Delivery** - For website owners who have visitors in multiple geographic locations, content will be delivered faster to these users as there is less distance to travel.
* **100 Percent Availability** - CDN users also benefit from the ability to easily scale up and down much more easily due to traffic spikes.

**3: Name some advantages of using CDN for static JS files and assets?**

* **It increases the parallelism available**. (Most browsers will only download 3 or 4 files at a time from any given site.)
* **It increases the chance that there will be a cache-hit**. (As more sites follow this practice, more users already have the file ready.)
* **It ensures that the payload will be as small as possible.** (Google can pre-compress the file in a wide array of formats (like GZIP or DEFLATE). This makes the time-to-download very small, because it is super compressed and it isn't compressed on the fly.)
* **It reduces the amount of bandwidth used by your server.** (Google is basically offering free bandwidth.)
* **It ensures that the user will get a geographically close response**. (Google has servers all over the world, further decreasing the latency.)
* (**Optional**) They will automatically keep your scripts up to date. (If you like to "fly by the seat of your pants," you can always use the latest version of any script that they offer. These could fix security holes, but generally just break your stuff.)

**4: What is a CDN origin server?**

The **origin server** is the primary source of your website's data and where your website files are hosted.

For example, if you are using DigitalOcean to host your site's files and have chosen data centre in San Francisco, then your origin server would be based in San Francisco.

Being limited to one server to deliver files from can be quite inefficient as distance is a contributing factor to latency.

**5: What is Azure CDN (Content Delivery Network) and why to use it?**

Almost each and every site has HTML, JS, CSS, image, video and font files that do not change much from user to user. To minimize the application load time we should use CDN.

**Azure Content Delivery Network (CDN) i**s CDN service provided by Azure Cloud Platform that enables in storing and accessing data on different content servers and locations – used by online or cloud services. It provides:

* Better performance and user experience for end-users who are far from the destination content source or where there are many intermediary nodes in between to reach the content.
* Ability to scale to handle instantaneous high load and demand
* Caching content from publicly available Azure storage blobs
* Caching web content, images, scripts, and other website content from CDN
* Enables in caching objects to the CDN that are provided by an Azure cloud service
* Cache content based on specific query strings
* Accessing cached content from a custom domain name by mapping the CDN HTTP Endpoint.

**6: What are CDN edge servers?**

Edge servers are the CDN servers used to cache content retrieved from your origin server or storage cluster. Another term often closely related to edge server is point of presence (POP). A POP refers to the physical location of where the edge servers are located.

That POP can have multiple edge servers caching content at that location.

Being able to deliver parts of a website from various locations helps decrease the distance between the visitor and the web server, thus reducing latency. This is exactly what CDN edge servers achieve.

**7: Why and how to use Cache Busting?**

Cache busting is useful because it allows your visitors to receive the most recently updated files without having to perform a hard refresh or clear their browser cache. From a developer's point of view, using cache busting is beneficial so that the latest changes can be pushed out and become available to everyone immediately. There are a few methods you can use to take advantage of cache busting:

* File name versioning (e.g. style.v2.css)
* File path versioning (e.g. /v2/style.css)
* Query strings (e.g. style.css?ver=2)

Both file name versioning and file path versioning are recommended cache busting methods. They do not interfere with any caching mechanisms and can be easily updated to reflect a modified file.

If using either file name or file path versioning there will be nothing to configure when using cache busting in conjunction with a CDN.

**8: What Is Cache Busting?**

When a static file gets cached it can be stored for very long periods of time before it ends up expiring. This can be an annoyance in the event that you make an update to a site however, since the cached version of the file is stored in your visitors' browsers, they may be unable to see the changes made. This is due to the fact that a visitor's browser will locally store a cached copy of your static assets given that your website is configured to leverage browser caching.

**Cache busting** solves the browser caching issue by using a unique file version identifier to tell the browser that a new version of the file is available. Therefore the browser doesn't retrieve the old file from cache but rather makes a request to the origin server for the new file.

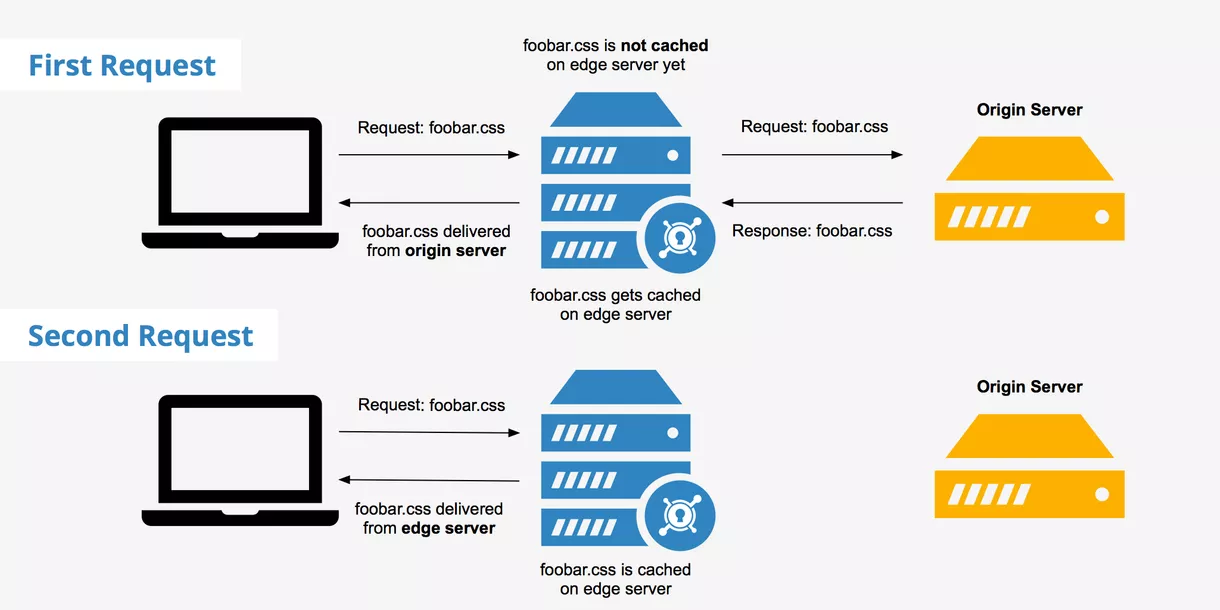
**9: How does CDN caching work?**

In the case of a CDN, the edge servers are where the data is cached and stored. Once you have integrated a CDN to work with your website, caching takes place as follows:

A visitor in a particular location (e.g. Chicago) makes the first request for a static asset on your site (e.g. style.css)

The asset is retrieved from your origin server and upon being delivered, the asset is cached on the KeyCDN Chicago edge server (i.e. the nearest KeyCDN POP based on that visitor's location).

If the same visitor makes a request for the same asset again, the request goes to the CDN POP edge server(s) to check if the asset is already cached. If the request hits an edge server that already has the asset cached, the visitor receives a response from that edge server. On the other hand, if the request hits a different edge server which doesn't have the asset cached yet, Step 2 is repeated.



Once your static assets are cached on all the edge servers for a particular location, all subsequent visitor requests for static assets will be delivered from the edge servers instead of the origin, thus reducing origin load and improving scalability.

**10: Name some advantages and disadvantages of Azure CDN**

**Advantage of Azure CDN**

* Better performance and improved user experience.
* Decrease the load on Original server because files are delivered from the edge server (CDN).
* Only configuration required to enable feature.
* Robust and less maintenance.
* Easy to configure and from resource only.

**Disadvantage of Azure CDN**

* You need to spend around $0.10 per GB. It doesn’t seem much but comparing it with your regular hosting plan bandwidth, It is too costly.
* Storing some sensitive files on the CDN network may open potential security vulnerabilities. Because it does not matter what you store at CDN, it will be copied to all distributed servers.
* Using CDN, you are creating an additional “point of failure”. If the CDN network goes down you may lose website visibility.

**11: Explain why CDN (in)availability may be a problem for using WebSockets?**

Today (almost 4 years later), web scaling involves using Content Delivery Network (CDN) front ends, not only for static content (html,css,js) but also your (JSON) application data.

Of course, you won't put all your data on your CDN cache, but in practice, a lot of common content won't change often. I suspect that 80% of your REST resources may be cached... Even a one minute (or 30 seconds) CDN expiration timeout may be enough to give your central server a new live, and enhance the application responsiveness a lot, since CDN can be geographically tuned...

To my knowledge, there is no WebSockets support in CDN yet, and I suspect it would never be. WebSockets are statefull, whereas HTTP is stateless, so is much easily cached. In fact, to make WebSockets CDN-friendly, you may need to switch to a stateless RESTful approach... which would not be WebSockets any more.

# Caching

**1: What is Caching?**

In computing, a cache is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data’s primary storage location. Caching allows you to efficiently reuse previously retrieved or computed data.

**2: Is Redis just a cache?**

Like a cache Redis offers:

* in memory key-value storage

But unlike a cash Redis:

* Supports multiple datatypes (strings, hashes, lists, sets, sorted sets, bitmaps, and hyperloglogs)
* It provides an ability to store cache data into physical storage (if needed).
* Supports pub-sub model
* Redis cache provides replication for high availability (master/slave)
* Supports ultra-fast lua-scripts. Its execution time equals to C commands execution.
* Can be shared across multiple instances of the application (instead of in-memory cache for each app instance)

**3: What is Resultset Caching?**

**Resultset caching** is storing the results of a database query along with the query in the application. Every time a web page generates a query, the applications checks whether the results are already cached, and if they are, pulls them from an in-memory data set instead. The application still has to render the page.

**4: What is Cache Invalidation?**

There are only two hard things in Computer Science: cache invalidation and naming things.

– Phil Karlton

HTTP caching is a solution for improving the performance of your web application. For lower load on the application and fastest response time, you want to cache content for a long period (TTL). But at the same time, you want your clients to see fresh (validate the freshness) content as soon as there is an update.

**Cache invalidation** gives you the best of both worlds: you can have very long TTLs, so when content changes little, it can be served from the cache because no requests to your application are required. At the same time, when data does change, that change is reflected without delay in the web representations.

**5: What usually should be cached?**

The results for the following processes are good candidates for caching:

* Long-running queries on databases,
* high-latency network requests (for external APIs),
* computation-intensive processing

**6: Name some Cache Writing Strategies**

There are two common strategies to write data in a cache:

1. **Pre-caching data**, for small pieces of data, usually during the application initialization, before any request.
2. **On-demand**, checking first if the requested data is in the cache (if the data is found, it is called a cache hit), using it, improving the performance of the application. Whenever the requested data has not been written to the cache (cache miss), the application will need to retrieve it from the slower source, then writing the results in the cache, thus saving time on subsequent requests for the same data.

**7: What are some alternatives to Cache Invalidation?**

There are three alternatives to cache invalidation.

* The first is to expire your cached content quickly by reducing its time to live (TTL). However, short TTLs cause a higher load on the application because content must be fetched from it more often. Moreover, reduced TTL does not guarantee that clients will have fresh content, especially if the content changes very rapidly as a result of client interactions with the application.
* The second alternative is to validate the freshness of cached content at every request. Again, this means more load on your application, even if you return early (for instance by using HEAD requests).
* The last resort is to not cache volatile content at all. While this guarantees the user always sees changes without delay, it obviously increases your application load even more.

**8 : Name some Cache Invalidation methods**

**Purge** - Removes content from cache immediately. When the client requests the data again, it is fetched from the application and stored in the cache. This method removes all variants of the cached content.

**Refresh** - Fetches requested content from the application, even if cached content is available. The content previously stored in the cache is replaced with a new version from the application. This method affects only one variant of the cached content.

**Ban** - A reference to the cached content is added to a blacklist (or ban list). Client requests are then checked against this blacklist, and if a request matches, new content is fetched from the application, returned to the client, and added to the cache. This method, unlike purge, does not immediately remove cached content from the cache. Instead, the cached content is updated after a client requests that specific information.

**9: Compare caching at Business Layer vs Caching at Data Layer**

* **Caching on the DAL** is straightforward and simple. Data access and persistence/storage layers are irresistibly natural places for caching. They're doing the I/Os, making them handy, easy place to insert caching. I daresay that almost every DAL or persistence layer will, as it matures, be given a caching function-if it isn't designed that way from the very start. Caching at the DAL/persistence layer risk having the "cold" reference data sitting there, pointlessly occupying X mb of cache and displacing some information that will, in fact, be intensively used in just a minute. Even the best cache managers are dealing with scant knowledge of the higher level data structures and connections, and little insight as to what operations are coming soon, so they fall back to guesstimation algorithms.
* **Caching in the business** is flexible and potentially more efficient. Application or business-layer caching requires inserting cache management operations or hints in the middle of other business logic, which makes the business code more complex. But the tradeoff is: Having more knowledge of how macro-level data is structured and what operations are coming up, it has a much better opportunity to approximate optimal ("clairvoyant" or "Bélády Min") caching efficiency.

Whether inserting cache management responsibility into business/application code makes sense is a judgment call, and will vary by applications. Lower complexity encourages higher correctness and reliability, and faster time-to-market. That is often considered a great tradeoff - less perfect caching, but better-quality, more timely business code.

**10: What are Cache Replacement (or Eviction Policy) algorithms?**

In computing, **cache algorithms** (also frequently called cache replacement algorithms or cache replacement policies or cache eviction policies) are optimizing instructions, or algorithms, that a computer program or a hardware-maintained structure can utilize in order to manage a cache of information stored on the computer. Caching improves performance by keeping recent or often-used data items in memory locations that are faster or computationally cheaper to access than normal memory stores. **When the cache is full, the algorithm must choose which items to discard to make room for the new ones.**

**Q11: What are some disadvantages of Cache Invalidation?**

**Invalidating** cached web representations when their underlying data changes can be very simple. For instance, invalidate /articles/123 when article 123 is updated. However, data usually is represented not in one but in multiple representations. Article 123 could also be represented on the articles index ( /articles), the list of articles in the current year (/articles/current) and in search results ( /search?name=123). In this case, when article 123 is changed, a lot more is involved in invalidating all of its representations. In other words, invalidation adds a layer of complexity to your application.

**Summary**:

* Using invalidation to transfer new content can be difficult when invalidating multiple objects.
* Invalidating multiple representations adds a level of complexity to the application.
* Cache invalidation must be carried out through a caching proxy; these requests can impact performance of the caching proxy, causing information to be transferred at a slower rate to clients

**12: Why is Cache Invalidation considered difficult?**

The non-determinism is why cache invalidation are unique and intractably hard problem in computer science.

* Computers can perfectly solve deterministic problems. But they can’t predict when to invalidate a cache because, ultimately, we, the humans who design and build computational processes, shall agree on when a cache needs to be invalidated. The hard and unsolvable problem becomes: how up-to-date do you really need data to be and when to change (or remove) it?
* Another problem a cache is (often by nature) much smaller compared to the overall amount of data that needs to be stored and if you just keep adding and adding elements to your cache, it becomes a full copy of your data. Respectively, you run out of memory quickly.

Basically it's difficult to achieve a desirable balance between stale objects stinking up your cache, and frequent unnecessary refreshes of unchanged objects considering limitation of the cache size.

**3: What is the difference between Cache replacement vs Cache invalidation?**

Frequently, the cache has a fixed limited size. So, whenever you need to write in the cache (commonly, after a cache miss), you will need to determine if the data you retrieved from the slower source should or should not be written in the cache and, if the size limit was reached, what data would need to be removed from it. That process is called Cache replacement strategy (or policy). Some examples of Cache replacement strategies are:

* Least recently used (LRU) - Discards the least recently used items first.
* Least-frequently used (LFU) - Counts how often an item is needed. Those that are used least often are discarded first.

**Cache invalidation is** the process of determining if a piece of data in the cache should or should not be used to service subsequent requests. The most common strategies for cache invalidation are:

* **Expiration time,** where the application knows how long the data will be valid. After this time, the data should be removed from the cache causing a “cache miss” in a subsequent request;
* **Freshness caching verification,** where the application executes a lightweight procedure to determine if the data is still valid every time the data is retrieved. The downside of this alternative is that it produces some execution overhead;
* **Active application invalidation,** where the application actively invalidates the data in the cache, normally when some state change is identified.

**14: Explain what is Cache Stampede**

**A cache stampede (or cache miss storm)** is a type of cascading failure that can occur when massively parallel computingsystems with caching mechanisms come under very high load. This behaviour is sometimes also called dog-piling.

Under very heavy load, when the cached version of the page (or resource) expires, there may be sufficient concurrency in the server farm that multiple threads of execution will all attempt to render the content (or get a resource) of that page simultaneously.

To give a concrete example, assume the page in consideration takes 3 seconds to render and we have a traffic of 10 requests per second. Then, when the cached page expires, we have 30 processes simultaneously recomputing the rendering of the page and updating the cache with the rendered page.

Consider:

| function fetch(key, ttl) {  value ← cache\_read(key)  if (!value) {  value ← recompute\_value()  cache\_write(key, value, ttl)  }  return value  } |
| --- |

If the function recompute\_value() takes a long time and the key is accessed frequently, many processes will simultaneously call recompute\\_value() upon expiration of the cache value.

In typical web applications, the function recompute\_value() may query a database, access other services, or perform some complicated operation (which is why this particular computation is being cached in the first place). When the request rate is high, the database (or any other shared resource) will suffer from an overload of requests/queries, which may in turn cause a system collapse.

**15: When to use LRU vs LFU Cache Replacement algorithms?**

* **LRU (Least Recently Used)** is good when you are pretty sure that the user will more often access the most recent items, and never or rarely return to the old ones. An example: a general usage of an e-mail client. In most cases, the users are constantly accessing the most recent mails. They read them, postpone them, return back in a few minutes, hours or days, etc. They can find themselves searching for a mail they received two years ago, but it happens less frequently than accessing mails they received the last two hours.
* **On the other hand, LRU ma**kes no sense in the context where the user will access some items much more frequently than others. An example: I frequently listen to the music I like, and it can happen that on 400 songs, I would listen the same five at least once per week, while I will listen at most once per year 100 songs I don't like too much. In this case, LFU (Least Frequently Used) is much more appropriate.

**16: What are best practices for caching paginated results whose ordering/properties can change?**

It seems what you need is a wrapper for all the parameters that define a page (say, pageNumber, pageSize, sortType, totalCount, etc.) and use this DataRequest object as the key for your caching mechanism. From this point you have a number of options to handle the cache invalidation:

* Implement some sort of timeout mechanism (TTL) to refresh the cache (based on how often the data changes).
* Have a listener that checks database changes and updates the cache based the above parameters (data refresh by server intent).
* If the changes are done by the same process, you can always mark the cache as outdated with every change and check this flag when a page is requested (data refresh by client intent).

The first two might involve a scheduler mechanism to trigger on some interval or based on an event. The last one might be the simpler if you have a single data access point. Lastly, it can quickly become an overly complicated algorithm that outweighs the benefits, so be sure the gain in performance justify the complexity of the algorithm.

**17: Cache miss-storm: Dealing with concurrency when caching invalidates for high-traffic sites**

**Problem**

For a high traffic website, there is a method (say getItems()) that gets called frequently. To prevent going to the DB each time, the result is cached. However, thousands of users may be trying to access the cache at the same time, and so locking the resource would not be a good idea, because if the cache has expired, the call is made to the DB, and all the users would have to wait for the DB to respond. What would be a good strategy to deal with this situation so that users don't have to wait?

**Answer**

The problem is the so-called Cache miss-storm (Cache Stampede or Dogpile) - a scenario in which a lot of users trigger regeneration of the cache, hitting in this way the DB.

To prevent this, first you have to set soft and hard expiration date. Lets say the hard expiration date is 1 day, and the soft 1 hour. The hard is one actually set in the cache server, the soft is in the cache value itself (or in another key in the cache server). The application reads from cache, sees that the soft time has expired, set the soft time 1 hour ahead and hits the database. In this way the next request will see the already updated time and won't trigger the cache update - it will possibly read stale data, but the data itself will be in the process of regeneration.

Next point is: you should have procedure for cache warm-up, e.g. instead of user triggering cache update, a process in your application to pre-populate the new data.

The worst case scenario is e.g. restarting the cache server, when you don't have any data. In this case you should fill cache as fast as possible and there's where a warm-up procedure may play vital role. Even if you don't have a value in the cache, it would be a good strategy to "lock" the cache (mark it as being updated), allow only one query to the database, and handle in the application by requesting the resource again after a given timeout.

**18: Name some Cache Stampede mitigation techniques**

**Answer**

**Locking** - a process will attempt to acquire the lock for the cache key and recompute it only if it acquires it. If implemented properly, locking can prevent stampedes altogether, but requires an extra write for the locking mechanism. Apart from doubling the number of writes, the main drawback is a correct implementation of the locking mechanism which also takes care of edge cases including failure of the process acquiring the lock, tuning of a time-to-live for the lock, race-conditions, and so on.

**External recomputation** - This solution moves the recomputation of the cache value from the processes needing it to an external process. This approach requires one more moving part - the external process - that needs to be maintained and monitored. The recomputation of the external process can be triggered in different ways:

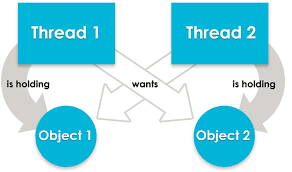
* When the cache value approaches its expiration
* Periodically
* When a process needing the value encounters a cache miss

**Probabilistic early expiration** - With this approach, each process may recompute the cache value before its expiration by making an independent probabilistic decision, where the probability of performing the early recomputation increases as we get closer to the expiration of the value. Since the probabilistic decision is made independently by each process, the effect of the stampede is mitigated as fewer processes will expire at the same time.

# Concurrency

**1: What is a Deadlock?**

* **A lock** occurs when multiple processes try to access the same resource at the same time. One process loses out and must wait for the other to finish.
* **A deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.



So, an example:

Resource A and resource B are used by process X and process Y

* X starts to use A.
* X and Y try to start using B
* Y 'wins' and gets B first
* now Y needs to use A
* A is locked by X, which is waiting for Y

| Thread 1 Thread 2  Lock1->Lock(); Lock2->Lock();  WaitForLock2(); WaitForLock1(); <-- Oops! |
| --- |

The best way to avoid deadlocks is to avoid having processes cross over in this way. Reduce the need to lock anything as much as you can. In databases avoid making lots of changes to different tables in a single transaction, avoid triggers and switch to optimistic/dirty/nolock reads as much as possible.

**2: Explain the difference between Asynchronous and Parallel programming?**

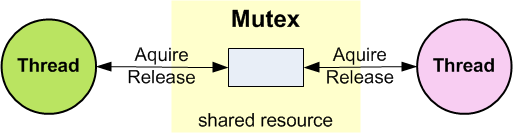
When you run something **asynchronously** it means it is non-blocking, you execute it without waiting for it to complete and carry on with other things. Parallelism means to run multiple things at the same time, in parallel. Parallelism works well when you can separate tasks into independent pieces of work. Async and Callbacks are generally a way (tool or mechanism) to express concurrency i.e. a set of entities possibly talking to each other and sharing resources.

Take for example rendering frames of a 3D animation. To render the animation takes a long time so if you were to launch that render from within your animation editing software you would make sure it was running asynchronously so it didn't lock up your UI and you could continue doing other things. Now, each frame of that animation can also be considered as an individual task. If we have multiple CPUs/Cores or multiple machines available, we can render multiple frames in parallel to speed up the overall workload.

**3: What is a Mutex?**

A Mutex is a mutually exclusive flag. It acts as a gate keeper to synchronise two threads. When you have two threads attempting to access a single resource, the general pattern is to have the first block of code attempting access, to set the mutex before entering the code. When the second code block attempts access, it sees that the mutex is set and waits until the first block of code is complete (and un-sets the mutex), then continues.

Specific details of how this is accomplished obviously varies greatly by programming language.



**4: Is there any difference between a Binary Semaphore and Mutex?**

A mutex (or Mutual Exclusion Semaphores) is a locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore (or Binary Semaphore) is signaling mechanism (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend called you, an interrupt will be triggered upon which an interrupt service routine (ISR) will signal the call processing task to wakeup. A binary semaphore is NOT protecting a resource from access. Semaphores are more suitable for some synchronization problems like producer-consumer.

Short version:

* A mutex can be released only by the thread that had acquired it.
* A binary semaphore can be signaled by any thread (or process).

**5: What is a Race Condition?**

A race condition is a situation on concurrent programming where two concurrent threads or processes compete for a resource and the resulting final state depends on who gets the resource first.

Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

| if (x == 5) // The "Check"  {  y = x \* 2; // The "Act"  // If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,  // y will not be equal to 10.  } |
| --- |

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock (Mutex or Semaphores) around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

| / Obtain lock for x  if (x == 5)  {  y = x \* 2; // Now, nothing can change x until the lock is released.  // Therefore y = 10  }  // release lock for x |
| --- |

**6: Write a function that guarantees to never return the same value twice**

**Problem**

Write a function that is guaranteed to never return the same value twice. Assume that this function will be accessed by multiple machines concurrently.

**Answer**

1. Just toss a simple (threadsafe) counter behind some communication endpoint:

| long x = long.MinValue;  public long ID(){  return Interlocked.Increment(ref x);  } |
| --- |

1. Let interviewer be the one that follow up with those problems:

* Does it need to survive reboots?
* What about hard drive failure?
* What about nuclear war?
* Does it need to be random?
* How random?

1. If they made it clear that it has to be unique across reboots and across different machines, I'd give them a function that calls into the standard mechanism for creating a new GUID, whatever that happens to be in the language being used. This is basically the problem that guids solve. Producing a duplicate Guid, no matter its format, is the most difficult lottery on the planet.

**7: Explain Deadlock to 5 years old**

Jack and Jill share a sparse kitchen that has only one of everything. They both want to make a sandwich at the same time. Each needs a slice of bread and each needs a knife, so they both go to get the loaf of bread and the knife from the kitchen.

Jack gets the knife first, while Jill gets the loaf of bread first. Now Jack tries to find the loaf of bread and Jill tries to find the knife, but both find that what they need to finish the task is already in use. If they both decide to wait until what they need is no longer in use, they will wait for each other forever. Deadlock.

**8: What is the meaning of the term “Thread-Safe”?**

**Problem**

Does it mean that two threads can't change the underlying data simultaneously? Or does it mean that the given code segment will run with predictable results when multiple threads are executing that code segment?

**Answer**

* Thread-safe code is code that will work even if many Threads are executing it simultaneously within the same process. This often means, that internal data-structures or operations that should run uninterrupted are protected against different modifications at the same time.
* Another definition may be like - a class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronisation or other coordination on the part of the calling code.

**9: How much work should I place inside a lock statement?**

It is first and foremost a question of correctness. You should not care so much about efficiency trade-offs, but more about correctness.

* Do as little work as possible while locking a particular object. Locks that are held for a long time are subject to contention, and contention is slow. Note that this implies that the total amount of code in a particular lock and the total amount of code in all lock statements that lock on the same object are both relevant.
* Have as few locks as possible, to make the likelihood of deadlocks (or livelocks) lower.
* If you want concurrency, use processes as your unit of concurrency. If you cannot use processes then use application domains. If you cannot use application domains, then have your threads managed by the Task Parallel Library and write your code in terms of high-level tasks (jobs) rather than low-level threads (workers).
* Benchmarking and being aware that there are more options than "lock everything everywhere" and "lock only the bare minimum".

**10: What is the difference between Concurrency and Parallelism?**

**Concurrency** is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

**Parallelism** is when tasks literally run at the same time, e.g., on a multicore processor.

For instance a bartender is able to look after several customers while he can only prepare one beverage at a time. So he can provide concurrency without parallelism.

**11: What's the difference between Deadlock and Livelock?**

* **In concurrent computing,** a deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock
* A livelock is similar to a deadlock, except that the states of the processes involved in the **livelock constantly** change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action,

**12: What are some advantages of Lockless Concurrency?**

**Lockless concurrency** eliminates deadlocks and provides the nice advantage that readers never have to wait for other readers. This is especially useful when many threads will be reading data from a single source.

**Lockless programming,** is a set of techniques for safely manipulating shared data without using locks. There are lockless algorithms available for passing messages, sharing lists and queues of data, and other tasks. Lockless programming is pretty complicated. e.g. All purely functional data structures are inherently lock-free, since they are immutable.

**13: What is Green Thread?**

**A Green Thread** is a thread that is scheduled by a virtual machine (VM) instead of natively by the underlying operating system. Green threads emulate multithreaded environments without relying on any native OS capabilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support.

Green Thread usually used when the OS does not provide a thread API, or it doesn't work the way you need. Thus, the advantage is that you get thread-like functionality at all. The disadvantage is that green threads can't actually use multiple cores. In the context of Java specifically, there were a few early JVMs that used green threads (IIRC the Blackdown JVM port to Linux did), but nowadays all mainstream JVMs use real threads. There may be some embedded JVMs that still use green threads.

**14: What is a Data Race?**

A data race happens when there are two memory accesses in a program where both:

* target the same location
* are performed concurrently by two threads
* are not reads
* are not synchronization operations

**15: What is Starvation?**

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads or threads with more "prioroty". For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

One more real live example may be this one. Imagine you're in a queue to purchase food at a restaurant, for which pregnant women have priority. And there's just a whole bunch of pregnant women arriving all the time. You'll soon be starving.

**16: Explain what is a Race Condition to 5 years old**

* You are planning to go to a movie at 5 pm. You inquire about the availability of the tickets at 4 pm. The representative says that they are available.
* You relax and reach the ticket window 5 minutes before the show. I'm sure you can guess what happens: it's a full house.

The problem here was in **the duration between the check and the action**. You inquired at 4 and acted at 5. In the meantime, someone else grabbed the tickets. That's a race condition - specifically a "check-then-act" scenario of race conditions.

**17: Provide some real-live examples of Livelock**

* A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.
* Example of livelock will happen where a husband and wife are trying to eat soup, but only have one spoon between them. Each spouse is too polite, and will pass the spoon if the other has not yet eaten.
* Deadlock detection may cause livelock. If two threads detect a deadlock, and try to "step aside" for each other, without care they will end up being stuck in a loop always "stepping aside" and never managing to move forwards.

**18: Two customers add a product to the basket in the same time whose the stock was only one (1). What will you do?**

**Problem**

What is the best practice to manage the case where two customers add in the same time a product whose the stock was only 1?

**Answer**

There is no perfect answer for this question and all depends on details but you have some options:

1. As a first 'defense line' I would try to avoid such situations at all, by simply not selling out articles that low if any possible.
2. You reserve an item for the customer for a fix time say (20 minutes) after they have added it to the basket – after they time they have to recheck the stock level or start again. This is often used to ticket to events or airline seats
3. For small jobs the best way is to do a final check right before the payment, when the order is actually placed. In worst case you have to tell you customer that you where running out of stock right now and offer alternatives or discount coupon.
4. Try to fulfil both orders later - just cause you don't have stock right now, doesn't mean you can't find it in an emergency. If you can't then someone has to contact the user who lucked out and apologise.

**Side note:**

1. The solution to do the double check when adding something to the basket isn't very good. People put a lot in baskets without ever actually placing an order. So this may block this article for a certain period of time.

**19: Compare Actor Model with Threading Model for concurrency**

* The actor model operates on message passing. Individual processes (actors) are allowed to send messages asynchronously to each other.
* What distinguishes this from what we normally think of as the threading model, is that there is (in theory at least) no shared state. And if one believes (justifiably, I think) that shared state is the root of all evil, then the actor model becomes very attractive.

**20: What happens if you have a "race condition" on the lock itself?**

**Problem**

For example, two different threads, perhaps in the same application, but running on different processors, try to acquire a lock at the exact same time. What happens then? Is it impossible, or just plain unlikely?

**Answer**

Have a "race condition" on the lock itself is impossible. It can be implemented in different ways, e.g., via the Compare-and-swap where the hardware guarantees sequential execution. It can get a bit complicated in presence of multiple cores or even multiple sockets and needs a complicated protocol (MESI protocol) between the cores, but this is all taken care of in hardware.

Compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

**21: What is the difference between Race Condition and Data Races? Are they the same?**

1. A data race occurs when 2 instructions from different threads access the same memory location, at least one of these accesses is a write and there is no synchronization that is mandating any particular order among these accesses.
2. A race condition is a semantic error. It is a flaw that occurs in the timing or the ordering of events that leads to erroneous program behavior. Many race conditions can be caused by data races, but this is not necessary.

The Race Condition and Data Races are not the same thing. They are not a subset of one another. They are also neither the necessary, nor the sufficient condition for one another.