FullStack.Cafe - Kill Your Tech Interview

Q1: What Is Load Balancing? ☆

Topics: Software Architecture Load Balancing

Answer:

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

1. Round robin
2. Session affinity or sticky session
3. IP Address affinity

Q2: What Is CAP Theorem? ☆

Topics: Software Architecture CAP Theorem

Answer:

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.

Q3: What Is Scalability? ☆☆

Topics: Software Architecture Scalability

Answer:

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.

Q4: What Is A Cluster? ☆☆

Topics: Software Architecture

Answer:

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.

Q5: Why Do You Need Clustering? ☆☆

Topics: Software Architecture

Answer:

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.

Q6: What is *Test Driven Development*? ☆☆

Topics: Agile & Scrum Software Architecture

Answer:

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**.

Q7: What is meant by the KISS principle? ☆☆

Topics: Software Architecture

Answer:

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.

Q8: What defines a software architect? ☆☆

Topics: Software Architecture

Answer:

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.

Q9: Why is it a good idea for “lower” application layers not to be aware of “higher” ones? ☆☆

Topics: Software Architecture Layering & Middleware

Answer:

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.

Q10: Define Microservice Architecture ☆☆

Topics: Microservices Software Architecture

Answer:

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

Q11: What do you mean by *lower latency interaction*? ☆☆

Topics: WebSockets Software Architecture

Answer:

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.

Q12: Why use WebSocket over HTTP? ☆☆

Topics: WebSockets Software Architecture

Answer:

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...

Q13: What is Domain Driven Design? ☆☆

Topics: Software Architecture DDD

Answer:

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.

Q14: What does the expression “Fail Early” mean, and when would you want to do so? ☆☆

Topics: Software Architecture

Answer:

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*.

Q15: What Is Session Replication? ☆☆☆

Topics: Software Architecture

Answer:

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.

Q16: What is Back-Pressure? ☆☆☆

Topics: Availability & Reliability Software Architecture

Answer:

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.

Q17: What is Elasticity (in contrast to Scalability)? ☆☆☆

Topics: Scalability Software Architecture

Answer:

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.

Q18: What is the difference between Monolithic, SOA and Microservices Architecture? ☆☆☆

Topics: Microservices Software Architecture SOA

Answer:

- **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.

Q19: WebSockets vs Rest API for real time data? Which to choose? ☆☆☆

Topics: API Design Software Architecture 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?

Solution:

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.

Q20: What does *program to interfaces, not implementations* mean? ☆☆☆

Topics: Design Patterns Software Architecture

Answer:

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.

FullStack.Cafe - Kill Your Tech Interview

Q1: What Do You Mean By High Availability (HA)? ☆☆☆

Topics: Software Architecture Availability & Reliability

Answer:

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.

Q2: What Is Middle Tier Clustering? ☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

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.

Q3: What Is Sticky Session Load Balancing? What Do You Mean By "Session Affinity"? ☆☆☆

Topics: Software Architecture Load Balancing

Answer:

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.

Q4: What Is Load Balancing Fail Over? ☆☆☆

Topics: Software Architecture Load Balancing

Answer:

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.

Q5: What Is ACID Property Of A System? ☆☆☆

Topics: Software Architecture Databases

Answer:

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.

Q6: How Do You Update A Live Heavy Traffic Site With Minimum Or Zero Down Time? ☆☆☆

Topics: Software Architecture Availability & Reliability

Answer:

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

Q7: What does SOLID stand for? What are its principles? ☆☆☆

Topics: Software Architecture

Answer:

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.

Q8: How can you keep one copy of your utility code and let multiple consumer components use and deploy it? ☆☆☆

Topics: Software Architecture

Answer:

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

Q9: Name some Performance Testing metrics to measure ☆☆☆

Topics: Software Testing Software Architecture

Answer:

- **Response time** - Total time to send a request and get a response.
- **Wait time** - Also known as average latency, this tells developers how long it takes to receive the first byte after a request is sent.
- **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.
- **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.
- **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.
- **Concurrent users** - This the most common measure of load — how many active users at any point. Also known as load size.
- **Requests per second** - How many requests are handled.
- **Transactions passed/failed** - A measurement of the total numbers of successful or unsuccessful requests.
- **Throughput** - Measured by kilobytes per second, throughput shows the amount of bandwidth used during the test.
- **CPU utilization** - How much time the CPU needs to process requests. Memory utilization - How much memory is needed to process the request.

Q10: Name some Performance Testing best practices ☆☆☆

Topics: Software Testing Software Architecture

Answer:

- 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.

Q11: "People who like this also like... ". How would you implement this feature in an e-commerce shop? ☆☆☆

Topics: Software Architecture

Answer:

Q12: What are the DRY and DIE principles? ☆☆☆

Topics: Software Architecture

Answer:

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

Q13: Is it better to return NULL or empty values from functions/methods where the return value is not present? ☆☆☆

Topics: Software Architecture

Answer:

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.

Q14: Explain the Single Responsibility Principle (SRP)? ☆☆☆

Topics: Software Architecture

Answer:

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.

Q15: What is the difference between Concurrency and Parallelism? ☆☆☆

Topics: Software Architecture Concurrency

Answer:

- **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*.

Q16: What does it mean "System Shall Be Resilient"? ☆☆☆

Topics: Software Architecture Availability & Reliability

Answer:

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.

Q17: What is difference between fault tolerance and fault resilience? ☆☆☆

Topics: Software Architecture

Answer:

- **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.

Q18: What is Domain in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

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).

Q19: What is a Model in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

> 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.

Q20: What is the difference between DTOs and ViewModels in DDD? ☆☆☆

Topics: Software Architecture DDD

Answer:

- 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.

FullStack.Cafe - Kill Your Tech Interview

Q1: Is *Unit Of Work* equals *Transaction*? Or it is more than that? ☆☆☆

Topics: Design Patterns Software Architecture

Answer:

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.

Q2: In OOP, what is the difference between the *Repository Pattern* and a *Service Layer*? ☆☆☆

Topics: ASP.NET ASP.NET MVC Design Patterns Software Architecture ASP.NET Web API

Answer:

- **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.

Q3: What is *Bad Design*? ☆☆☆

Topics: Software Architecture

Answer:

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.

Q4: What is the difference between *Behavior-Driven Development (BDD)* vs *Domain-Driven Design (DDD)*? ☆☆☆

Topics: Software Architecture DDD

Answer:

- **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.

Q5: What is the *Command and Query Responsibility Segregation (CQRS)* Pattern? ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

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.

Q6: Name some benefits of *CQRS Pattern* ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

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.

Q7: Describe what is the *Event Sourcing Pattern* ☆☆☆

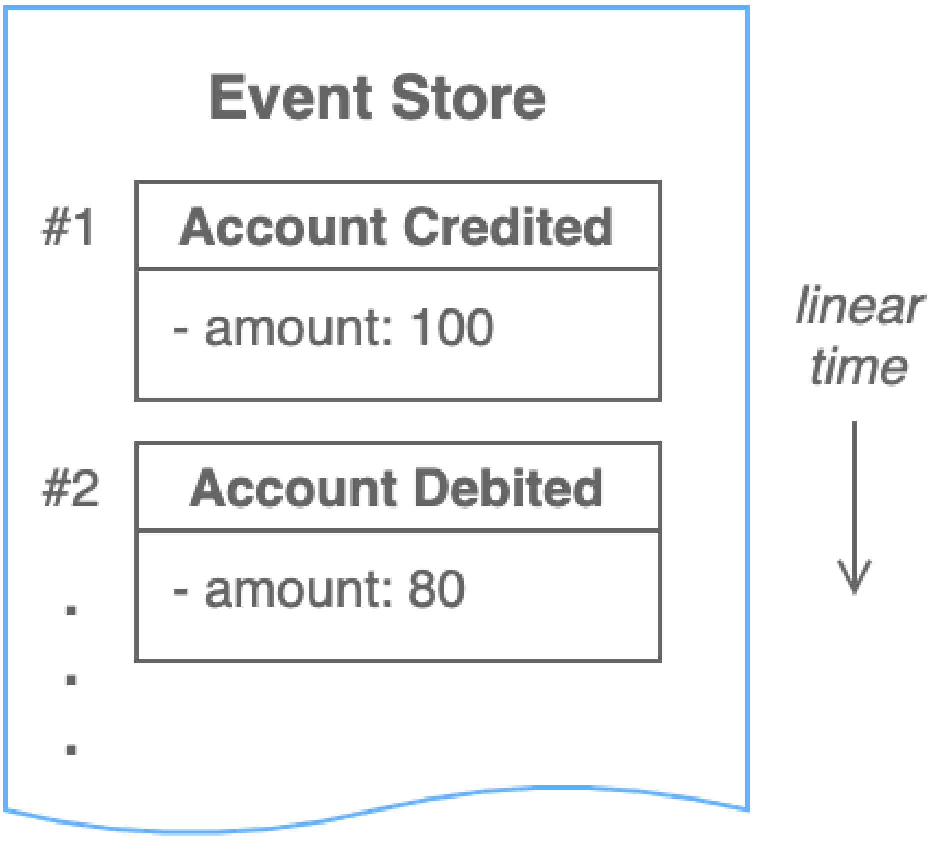
Topics: DDD Software Architecture Design Patterns

Answer:

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.



Q8: What Is IP Address Affinity Technique For Load Balancing? ☆☆☆☆

Topics: Software Architecture Load Balancing

Answer:

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.

Q9: What Is Sharding? ☆☆☆☆

Topics: Software Architecture Databases

Answer:

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.

Q10: What Is BASE Property Of A System? ☆☆☆☆

Topics: Databases Software Architecture NoSQL

Answer:

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.

Q11: Why should you structure your solution by components? ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

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...)

Q12: Why layering your application is important? Provide some bad layering example. ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

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

Q13: Explain threads to your grandparents ☆☆☆☆

Topics: Software Architecture

Answer:

Q14: How to handle exceptions in a layered application? ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

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);
        }
    }
}
```

Q15: What's the difference between principles YAGNI and KISS? ☆☆☆☆

Topics: Software Architecture

Answer:

- **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.

Q16: What is GOD class and why should we avoid it? ☆☆☆☆

Topics: Software Architecture

Answer:

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.

Q17: Provide Definition Of Location Transparency ☆☆☆☆

Topics: Software Architecture

Answer:

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.

Q18: Explain Failure in Contrast to Error ☆☆☆☆

Topics: Availability & Reliability Software Architecture

Answer:

- 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.

Q19: Why should I isolate my domain entities from my presentation layer? ☆☆☆☆

Topics: Software Architecture 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...

Solution:

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.

Q20: Two customers add a product to the basket in the same time whose the stock was only one (1). What will you do? ☆☆☆☆

Topics: Concurrency Software Architecture

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?

Solution:

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.

FullStack.Cafe - Kill Your Tech Interview

Q1: Defend the monolithic architecture. ☆☆☆☆

Topics: Software Architecture

Answer:

Q2: What is *Unit test*, *Integration Test*, *Smoke test*, *Regression Test* and what are the differences between them? ☆☆☆☆

Topics: Software Architecture Software Testing Unit Testing

Answer:

- **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.

- **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.

- **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.

- **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:

- **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.

- **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).

- **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.

- **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

Q3: What is actor model in context of a programming language? ☆☆☆☆

Topics: Software Architecture

Answer:

**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*.

Q4: How do you off load work from the Database? ☆☆☆☆

Topics: Databases Software Architecture

Answer:

Here is a list of standard options.

- **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.
- **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.
- 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.
- 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.
- **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.
- 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.
- 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.

Q5: Explain what is Cache Stampede ☆☆☆☆

Topics: Caching Software Architecture

Answer:

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.

Q6: Compare "Fail Fast" vs "Robust" approaches of building software ☆☆☆☆

Topics: Availability & Reliability Software Architecture

Answer:

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.

Q7: What will you choose: *Repository Pattern* or "smart" business objects? ☆☆☆☆

Topics: Design Patterns Software Architecture

Problem:

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?

Solution:

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.

Q8: Is *Repository Pattern* as same as *Active Record Pattern*? ☆☆☆☆

Topics: Design Patterns Software Architecture

Answer:

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.

Q9: How should I be *grouping* my Repositories when using Repository Pattern? ☆☆☆☆

Topics: Design Patterns Software Architecture

Answer:

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());
```

Q10: What is relationship between *Repository* and *Unit of Work*? ☆☆☆☆

Topics: Design Patterns Software Architecture Entity Framework

Answer:

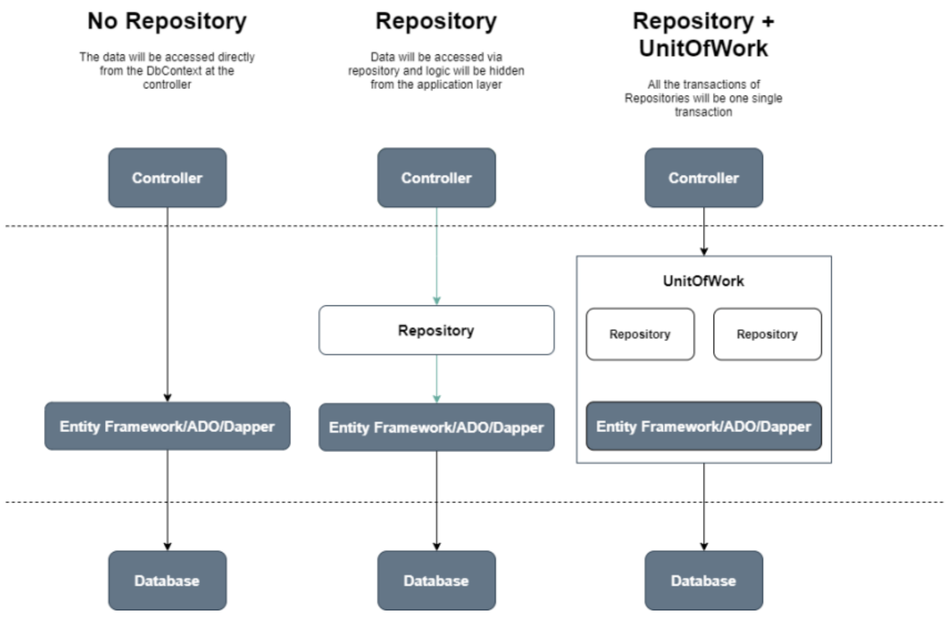
- The **Unit of Work** pattern "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." **UoW** orchestrates atomic data operations that span more than one repository. Think of a unit of work as the director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


No Repository
The data will be accessed directly from the DbContext at the controller
Repository
Data will be accessed via repository and logic will be hidden from the application layer
Repository + UnitOfWork
All the transactions of Repositories will be one single transaction
Controller
Controller
Controller
UnitOfWork
Repository
Repository
Repository
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Database
Database
Database

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

Q11: What is the *Dependency Inversion Principle (DIP)* and why is it important? ☆☆☆☆

Topics: Software Architecture Dependency Injection

Answer:

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.

Q12: What Does Eventually Consistent Mean? ☆☆☆☆☆

Topics: Software Architecture Databases

Answer:

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.

Q13: What Is Shared Nothing Architecture? How Does It Scale? ☆☆☆☆☆

Topics: Software Architecture

Answer:

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.

Q14: What are heuristic exceptions? ☆☆☆☆☆

Topics: Software Architecture

Answer:

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.

Q15: Are you familiar with The Twelve-Factor App principles? ☆☆☆☆☆

Topics: Software Architecture

Answer:

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.

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

Q16: Why is writing software difficult? What makes maintaining software hard? ☆☆☆☆☆

Topics: Software Architecture

Answer:

Q17: How do I *test* a `private` function or a class that has `private` methods, fields or inner classes? ☆☆☆☆☆

Topics: Software Architecture Unit Testing

Answer:

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.

Q18: What does Amdahl's Law mean? ☆☆☆☆☆

Topics: Software Architecture Reactive Systems

Answer:

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.

Q19: What is the difference between Cohesion and Coupling? ☆☆☆☆☆

Topics: Microservices Software Architecture

Answer:

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**.

Q20: What is the most accepted transaction strategy for microservices? ☆☆☆☆☆

Topics: Microservices Software Architecture

Answer:

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.

FullStack.Cafe - Kill Your Tech Interview

Q1: Could you provide an example of the *Single Responsibility Principle*? ☆☆☆☆

Topics: Software Architecture

Answer:

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.

Q2: What are best practices for caching paginated results whose ordering/properties can change? ☆☆☆☆☆

Topics: Caching Software Architecture

Answer:

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.

Q3: Cache miss-storm: Dealing with concurrency when caching invalidates for high-traffic sites ☆☆☆☆☆

Topics: Caching Software Architecture

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?

Solution:

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.

Q4: Where DTO should be implemented, in a *Domain Layer* or in an *Application Service Layer*? Explain. ☆☆☆☆☆

Topics: DDD Software Architecture Layering & Middleware

Answer:

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.

Q5: Can we use the *CQRS* without the *Event Sourcing*? ☆☆☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

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.