N-tier architecture

Wednesday, 10 January 2018 12:34 PM

N-tier architecture

- similar app with logical layers and physical tiers.
- layers are a way to separate responsibilities and manage dependencies.
- each layer has specificity.
- typical layers are: ui, business layer, logic and db. this also called 3-tier.

- Tiers are physically separate running on separate machines. A tier can call to another tier directly or use intermediate messaging.
- Correct layers matter in large, long-lived, business-critical systems.
- Physically separating the tier improves scalability and fault tolerance. Easier to understand network communication.
- N-tier architecture can be 1-tier with layers:
  - ui + business layer + data layer: can't run on next layer monolithic deploy.
  - ui + business layer + data layer: can run on next layer monolithic deploy.
- A class diagram with both the dependencies between layers. However it might still without layers, if the layer properly passes request along to the next layer.

Tiers

1-tier arch

- Traditional layers: user interactions are tightly coupled.
- Business logic layer: accepts the data from the ui layer, validates it to fit business logic.
- Data Access layer: accepts the data from the business layer and performs the necessary queries.

2-tier arch

- Traditional layers runs on the client and communicates with a db server.
- ui business logic layer: ui accepts the user input, validates the client side.

Single-tier - 3-tier arch

- complicated case, as it is equivalent to running 3-tier app on a personal comp.

Advantages of N-Tier

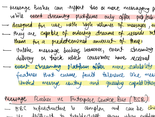
- Low response availability
- Better availability of layers when one layer is down.
- Upgrade this layer can be done from an server.
- Improve maintenance: ui developer people can manage ui layer.

Disadvantages

- increased complexity of ui as a whole.
- increased network latency as one of this layers
- upgrade an existing tier that will break the tier dependency and
- difficult to manage network security.

Message Buses

- software that enables application systems and services to communicate with each other and exchange information.
- messaging buses are like big transshipping warehouse between several messaging protocols. They allow interoperability between a wide range of applications, despite mismatches of language and underlying platforms.



- messaging business are middle, clear, which need follow messages to appropriate destinations. They serve as intermediaries between other apps, allowing them to communicate without knowing where the recipient app resides or what kind of software and hardware they run.
- The facilities for designing & processing and delivery certain messages.

Models

Message Bus: offer four basic message distribution patterns or messaging styles:

1) Point-to-point messaging
   - this is distribution pattern similar to queue, queue acts a mediator, distributing messages between two applications.
2) Publish-subscribe messaging
   - often referred as topic, publisher sends messages to topic, subscribers receive messages from topic.

Message Buses as Broker Messaging

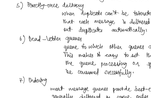
- messaging buses can support one or more messaging patterns, messaging queue and publish/sub, while other messaging patterns only offer one or two specific distribution patterns.
- designed for use with loosely coupled messaging, which decouples systems and enables them to communicate with one another without direct or closer coupling, called tight, and changing them to a predetermined messaging by hand.
- Unlike messaging buses, tightly coupled systems can offer synchronous messaging, delivery or block which consumes more resources for messaging.
- event driven messaging platform which more reliable than message buses but from platform that provide publish/subscribe messaging, as well as more, limited messaging delivery and guaranteed delivery.

Messaging Buses vs Messaging Queue Bus (MQB)

- B2B interoperability is complex and can be challenging to integrate and maintain in-house. It's difficult to establish connections with other business partners, or not easy to create, and updates by business.
- Message messaging buses are lightweight alternative to B2B that provide similar functionality as mechanisms for inter-enterprise communication of a business.
- well suited for non-critical business data and have lesser more expensive as B2B. have below both of factors.
- e.g. message buses: — NATS, Apache Kafka, RabbitMQ, ActiveMQ.

Message Queue

- Queue of events-to-process, communicates that facilitates asynchronous communication.
- It asynchronously receives messages from producers and sends them to consumers.
- Queues are used to decouple messaging senders and receivers, allowing them to operate independently.
- In cloud systems with microservices distributed hosts and remote clients, queues can be particularly good. Messages in queues and large systems which can span an almost worldwide network of hosts.



Benefits

- Decoupling: — messages are stored in the queue until they are processed and deleted.
- Each message is processed only once, by a single consumer.
- Ordering:
  - A producer publishes a job to the queue, then notifies the user of job status.
  - A consumer picks up the job from the queue, performs the work, and then marks the job as complete.

Advantages of message queue

- Scalability: — allow asynchronous jobs, multiple workers — add app can add and remove to the queue without affecting the rest of system.
- Decoupling: — message queue removes dependencies between components and significantly improves the maintainability and flexibility of the system.
- Performance: — enable improve communication between services that are independent that are performing and consuming messages in batch offer speed and not wait idle.
  - (performance improvement comes without waiting for them to be finished)
- Reliability: — queues enable data data persistence and replicate the queues that happen when delivered back or system go failure.

Synchronous

1) Push or Pull Delivery
   - Push message queues provide, both push and pull options for distributing messages.
   - Pull means consumers periodically poll the queue for new messages.
   - Push means that a consumer is notified when a message is available.
   - we can also use long-polling to allow polling a server a specified amount of time, for new messages to arrive.
2) FIFO Queues
   - Stand for (first) (out) (queue), consumers called head, is processed first.
3) Scheduled or Delay Delivery
   - Message queues have support a specific delivery time for a message, if we need to have a message delay for all messages, we can delay delay queue.
4) At-least-once-delivery
   - Many check multiple copies of messages for redundancy and high availability, and ensure messages in the event of communication failures or crash to ensure they are delivered at least once.
5) Twenty-once delivery
   - when duplicate can't be tolerated, FIFO message queues will make one that each message is delivered exactly once (and only once) by filtering out duplicate messages.
6) Dead-letter queues
   - queue, to which other queues can send messages that could be processed successfully. This enable it easy to see these messages that failed, further troubleshoot without blocking the queues, preventing the queuing API replay on a message that might cause the consumer crashing.
7) Ordering
   - most message queues provide best-effort ordering which ensures that messages are generally delivered in order, order is lost if there need a message to delivered at least once.
8) Poison-pill messages
   - Poison-pills are special messages that can be received but not processed. They are a malformed message or bugs in the consumer, so that the work is no longer working for user requests, and are similar to client in a client/server model.
9) Security
   - All data authorization, ppl that keep to access the queue, this allows us to encrypt messages since the queue, as well as on queue itself.
10) Trade-Offs
    - message broker most that also called jobs, can flow from different that's specific. They can support scheduling and can be used to run computationally-intensive jobs in the background.

Backpressure — If queue starts grow significantly, queue size can become large, then memory, resulting in cache misses, disk reads, and even cache frequency.
Backpressure can help by limiting the queue size (blocking enqueuing a new message until the oldest message has been dequeued). Or discarding messages if queue is full.
- Note, the queue fills up, clients get a queue busy or HTTP 503 unavailable to try again later.
- Queue can also support set a time limit, jobs expiration before processing.

e.g.: Amazon SQS, ElastiMQ, ActiveMQ, Google

Publish-Subscribe

- similar to message queues, publish is also a queue but, course-to-course communication that facilitates asynchronous communication.
- In publish model any message published to a topic is pushed immediately to all the subscribers of the topic:

- subscribers to the message topic often perform different functions and can each do something different with message in parallel.
- publisher doesn't need to know who is using the topic that it broadcasting and unknown consumer need to know where message comes from.
- This is different queues model, in the queue model that sends the message, often known the distribution to a single sub.

Messaging

Brokers, MQ, which handle messages sent from one application to another. Topics messages with URIs, so one producer and pushes those out immediately to all subscribers.
- messaging topic provides a lightweight mechanism to broadcast asynchronous event notifications that allows endpoints, independent of the sender to the topic, to subscribe and receive those messages.
- to broadcast a message, a component called publisher simply pushes a message to the topic.
- All components that subscribe to the topic (known as subscribers) will receive every message that was broadcasted.

Advantages

- Loosely coupling
  - message topic allows asynchronous, point-based delivery relationship exist for messaging. Guarantees the availability checks on pull for send info and updates.
  - this primarily facilitate response time and improve efficiency delivery.
- Asynchronous Throughput
  - enable delivering of services easier, more robust and less complex.
  - instead of establishing a path to point where one app can send messages, a publisher will simply push messages to a topic.
- Independent from change, approach availability to disperse and the system dependability efficient.
- Automated load balancing and scaling
  - publishers and subscribers are decoupled and scale independently from each other, allowing one to develop and scale from independently.
- Complexity communication
  - publish model enables complexity by requiring all pub-to-sub messaging with a single connection to a message broker, which will manage subscriptions and deliver messages should be delivered to which subscribers.

Features of publish model:

1) Push Delivery
   - publish messaging leadership pushes msgs, and distribution when messages are published to a message topic.
   - subscribers to message topic.
   - subscribers has offered when a message is available.
2) Multiple Delivery Protocols
   - topic can typically connect to multiple type of subscribers, such as mobile, browsers, HTTP servers etc.
3) Transient
   - this transient happens when a message is sent to a topic and then published and pushed to multiple subscribers.
   - transient provides msgs exact satisfaction which is more almost for parallel processing.
4) Filtering
   - this feature enhances the subscribers to receive a message filtering policy so that all msgs not fit the specifications to be subscribed for, is opposed to receiving every single message published to the topic.
5) Scalability
   - Publish messaging clients: often provide very high scalability and low latency delivery by sending copies of the same message to multiple clients.
6) Security
   - messaging topic authorization: ppl tied key to publish/subscribe, this allows us to use assigned authority and through messaging to transmit key the subscribers.

e.g.: publish — Amazon SNS — messaging topic.

Enterprise Service Bus (ESB)

- architectural pattern enhances a centralized software component performs integration between applications.
- It performs transformations of data, routing, message complexity, protocol conversion, message communication protocol, and potentially enhances the complexity of multiple services.
- The ESB can make these integrations and transformations available as a centralized interface, for client to use easily efficiently.

Advantages

- In theory, a centralized ESB offers the benefits of simplicity and availability complexity communication management and integration between services from the enterprise.
- Improved flexibility
  - enterprise standards to integrate, new technologies who are part of an app integration.
- Improve reuse, cost-effective, scalability
  - components can be created independently of those
- Greater efficiency
  - Enable, set, have, management app and broker the access, and each consumer can choose, to be more availability requirement without delay the availability of other components in the system.

Disadvantages

ESB can be error as bottleneck:
- making changes to enhancements to ESB integration could disproportionately affect one use, the core integration.
- Single point of failure: can bring down all communications.
- update to ESB often target specific independent of their ui integration requires to perform many updates.
- ESB is centrally managed which makes cross-team collaboration challenges
- tight integration and maintenance complexity.

e.g.: Azure Service Bus, IBM App Connect, Mulesoft Connect.

Monolithic and Microservices

A monolith is a self contained and independent application. It's built as a single unit and is responsible for user UI + database. Thus, you deploy one big monolithic unit.

Advantages

- Simple to develop — just one codebase, one language.
- Simple to test — end-to-end testing is straightforward.

Benefits

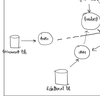
- Easier to understand — fewer moving parts, fewer dependencies.
- Easier to deploy — one artifact, one deployment.
- Easier to scale — horizontally by cloning the entire application.
- Easier to debug — all logs in one place, no distributed tracing needed.
- Simple transactions — ACID transactions work naturally.

Monolithic Problems

- As monolith grows, it becomes harder to understand, test, and deploy. A small change requires rebuilding and redeploying the entire application.
- Tight coupling — changes in one part affect other parts unexpectedly.
- Scaling limitations — can't scale individual components independently.
- Technology lock-in — stuck with the technology you started with.
- Team bottlenecks — multiple teams working on the same codebase cause conflicts.

Microservices

- Instead of thinking just one, consider which set of data, which is different aspects to deploy data.
- Each microservice has its own database, which is different from monolithic approach.
- Microservices are independent, deployable units, which allows to deploy independently.
- Good control in a certain phase of business logic, that provide an explicit boundary around which a domain model exist.



- each service has its separate database which can be managed by small development team.
- Course can be deployed independently and team can update, as anything needs without scalability and deployment for entire application.
- course are responsible for exactly their own data in external store (database or cache). This differs from traditional model where a separate database serves the data requests.

Characteristics

1) Loosely coupled
   - Each service can be independently deployed and scaled.
   - Leads to decentralization of development teams and thus isolating them to focused and agile teams with minimal overhead and operational dependencies.
2) Small but focused
   - Each task and responsibility, and size. Course should focus on one problem (business or core). This differs from traditional model where a separate database serves the data requests.
3) Built for business
   - Usually organized around business capabilities and products.
4) Validated and fault tolerance
   - designed to a step in further to use of failures or errors.
   - In microservice architecture independently deployable units, failed business is of impact isolation.
5) Highly maintainable
   - Could be easy to maintain and test

Adv

- Loosely coupled course
- course can be deployed independently
- Loosely coupled, fault tolerance frames
- Improve fault tolerance and data isolation
- Better availability as each course can be scaled independently
- Flexibility any programming language or technology stack.

Benefits

- Complexity of a distributed system
- Testing is more difficult
- Expensive to maintain (additional courses and db etc)
- Inter-course communication has its own challenges
- data integrity and consistency
- network complexity and latency

Microservice Best Practices

- model course around business capabilities
- courses should have their logging and logic functional cohesion
- update patterns and well, validating checksums to prevent failure cohesion & coupling from cascading
- courses should only communicate through well-designed APIs. Avoid leaking implementation details.
- each course should own its private, its course, that stores the data.
- avoid sharing database courses. Course is responsible having control job schemas and data manipulation protocols.
- Accountability complexity, distributed tracing are responsible for diagnosing and building courses. Avoid coupling code or data schemas.
- Fail fast: timing a check timeouts to achieve fault tolerance.
- Business, Dev Ops, API changes are backward compatible.

Drawbacks of microservices

- course boundaries are not based on the business domain.
- Underestimating how hard it is build a distributed system.
- Underestimate the complexity implementation integration.
- lack of business alignment
- lack of clear ownership
- lack of testing for fault tolerance, easy upset to cascading failures.

Examples of distributed monoliths

- distributed monoliths is a system that requires the coordination architecture, but is tightly coupled without flexibility, or a monolithic application adopting microservices architecture, course units a lot of interdependencies, that cause ripple effect, and, and, and