Chapter 1: Escaping Monolithic hell

Over the time an application developed as a monolith application (single war file) deployable war/jar file can lead to something called as monolithic hell. Which is largely recognized by an infamous design patter call as big ball of mud. To quote Foote and Yoder, the authors who coin the word big ball of mud states that, it's a "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle." The code delivery is slow as the code is written in an obsolete framework, adding a new business function take a long time, also introducing a framework or technology is almost impossible due to tight coupling nature of the application.

Monolithic hell

- Complex intimidates developer. Over the time, application code grows its complexity multifield making it difficult for a single developer to understand. There are multiple team, developing different component of the application coordination within those team is an daunting task.
- **Development is slow**: Due to large code base, it takes long time to load and start up on developer's development environment.
- **Build and Release process is cumbersome**, as there are multiple developers working in parallel, managing and merging change is cumbersome.
- **Scaling is difficult**: need large memory and CPU foot print to scale the application.
- **Lack of fault isolation**, fault in one module easily led to a failure of the entire application.
- **Moving away from obsolete tech stack is difficult even sometime impossible**, upgrading or changing tech stack is difficult.

Comparing Microservice Architecture with SOA Architecture

SOA Architecture	Microservice Architecture
In case of SOA, it's the smart pipes and dumb endpoints architecture where the SOA architecture are centrally managed by a enterprise service bus using heavyweight protocols like SOAP	In case of microservices, its more like a dumb pipes and smart endpoints where messages are transmitted using a message broker or a direct service-to-service communication.
Follow common data model	Data model, database per service
Large monolithic application.	Smaller microservices.

Key benefit of microservice

- Can be easily implement CI/CD pipeline.
- Services are small and manageable by a small team.
- Services are independently deployable.
- Services are independently scalable.
- Enable team to be autonomous, have de-centralized team & decisioning.
- Allows easier adaptation of new technologies.
- Better fault isolation.

Drawbacks of microservice

- Finding the right set of services is challenging.
- Distributed system are complex, it makes design & development, testing , deployment difficult.
- Deploying a feature that span across multiple services need higher degree of coordination.
- Deciding to adapting microservice architecture is difficult.

Chapter 2: Decomposition

What is a software architecture?

The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

4+1 Model
In 4+1 software architecture model define 4 different view of software architecture – each of the view describes a particular aspect of the architecture and consist of particular set of software element and their relationship between them.

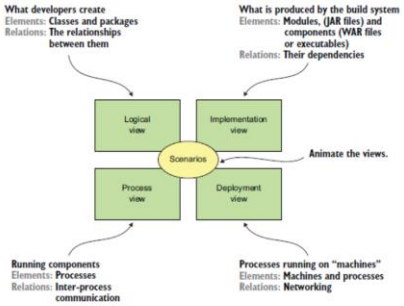


Figure 2.1 The 4+1 view model describes an application's architecture using four views, along with scenarios that show how the elements within each view collaborate to handle requests.

- **Logical view** — The software elements that are created by developers. In object-oriented-languages, these elements are classes and packages. The relations between them are the relationships between classes and packages, including inheritance, associations, and depends-on.
- **Implementation view**—The output of the build system. This view consists of modules, which represent packaged code, and components, which are executable or deployable units consisting of one or more modules. In Java, a module is a JAR file, and a component is typically a WAR file or an executable JAR file. The relations between them include dependency relationships between modules and composition relationships between components and modules.
- **Process view** — The components at runtime. Each element is a process, and the relations between processes represent inter process communication.
- **Deployment view** – How processes are mapped to a machine, the element in this view consists of the physical and virtual machines and the processes. The relation between the machine represents the networking. This view also describes the relationship between processes and machines.

  In addition to the four views (Scenario View) – there is also an additional view which describes, how various components within each view collaborate to handle a request.

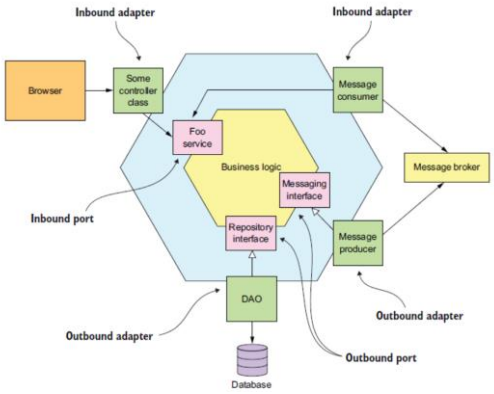
Why Software architecture matters?

There are two types of requirements – *functional* and *non-functional* requirements *also known as* – *ilities properties (scalabilities, reliabilities, accountabilities, maintainability etc)*, software architecture matter most for the non-functional requirements as much as it does for the functional requirements.

Layer Architecture Style: A layered architecture organizes software elements into layers. Each layer has a well-defined set of responsibilities. A layered architecture also constraints the dependencies between the layers. A layer can only depend on either the layer immediately below it (if strict layering) or any of the layers below it.

Hexagon Architecture Style: Hexagonal architecture is an alternative to the layered architectural style. Instead of the presentation layer, the application has one or more inbound adapters that handle requests from the outside by invoking the business logic. Similarly, instead of a data persistence tier, the application has one or more outbound adapters that are invoked by the business logic and invoke external applications. A key characteristic and benefit of this architecture is that the business logic does not depend on the adapters. Instead, they depend upon it.

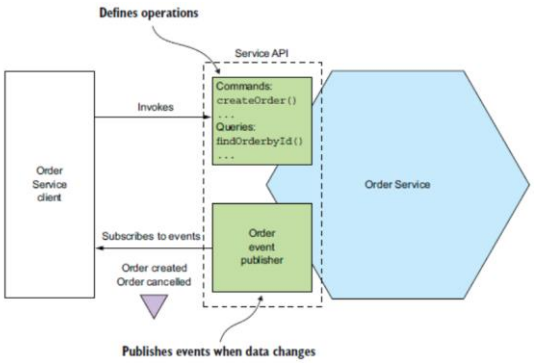
The business logic has one or more ports. A port defines a set of operations and is how the business logic interacts with what's outside of it. In Java, for example, a port is often a Java interface. There are two kinds of ports: inbound and outbound ports. An inbound port is an API exposed by the business logic, which enables it to be invoked by external applications. An example of an inbound port is a service interface, which defines a service's public methods.



An important benefit of the hexagonal architecture is the business logic is decoupled from presentation and data access layer. The business logic can be invoked from multiple entry points adapters.

Microservice Architecture Style: The microservice architecture is also an architectural style. It structures the implementation view as a set of multiple components: executables or WAR files.

Service*:* A service is a standalone, independently deployable software component that implements some useful functionality.



Share Libraries: Often the reusable code is packaged as shared libraries, modules that needs to access these functionalities – can use the same without any need to duplicating the logic. For example – Common Object will be used across multiple services – instead of defining it all projects we can have common Object define as shared libraries and use it across multiple projects.

Steps to decompose a microservices:

1. Step1 define domain model consisting of the key classes.
2. Step2 Define system operation and describe each one of the key classes behaviours in term of domain model.

System Behaviour:

- Domain model derives by the noun of the user stories, where system operations are mostly defined by the verbs of the user stories.
- User stories should be written using *Given – When – Then*
- System Behaviour – there are two types of system behaviours – *Command & Query*, which are exposed over system endpoint as REST or gRPC endpoint.

Each of the system behaviour should be define in the below format.

Operation	*Method ( ) name & signature*
Return	*Return value*
Pre-Condition	*Condition that fulfil given conditions of the User Stories*
Post-Conditions	*Conditions that fulfil then conditions of the User Stories*

<u>Examples:</u>
System Behaviour CreateOrder

Operation	`createOrder` (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)
Returns	`orderId`, ...
Preconditions	▪ The consumer exists and can place orders. ▪ The line items correspond to the restaurant's menu items. ▪ The delivery address and time can be serviced by the restaurant.
Post-conditions	▪ The consumer's credit card was authorized for the order total. ▪ An order was created in the `PENDING_ACCEPTANCE` state.

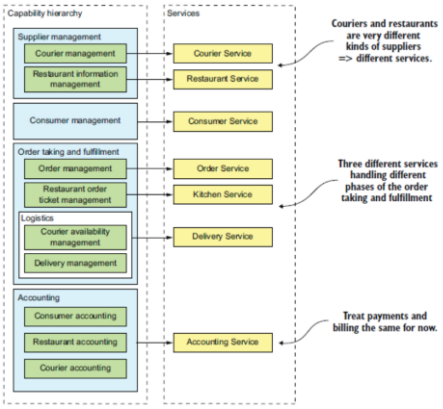
System Behaviour acceptOrder

Operation	`acceptOrder(restaurantId, orderId, readyByTime)`
Returns	—
Preconditions	▪ The `order.status` is `PENDING_ACCEPTANCE`. ▪ A courier is available to deliver the order.
Post-conditions	▪ The `order.status` was changed to `ACCEPTED`. ▪ The `order.readyByTime` was changed to the `readyByTime`. ▪ The courier was assigned to deliver the order.

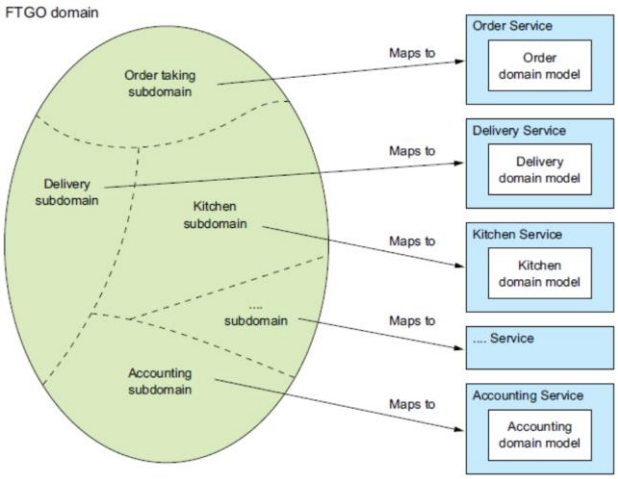
Decomposition Strategies:

Decomposition by Business Capabilities , Define business capabilities, business capabilities define collectively the Organization Business. Business capabilities can be further breakdown into services.

Organization Business > Business Capabilities > Services



Decomposing Services by applying Sub-Domain Pattern, DDD defines a separate domain model for each subdomain. A subdomain is a part of the domain, DDD's term for the application's problem space. Subdomains are identified using the same approach as demystify business capabilities: analyze the business and identify the different areas of expertise. The end result is very likely to be subdomains that are similar to the business capabilities. DDD calls the scope of a domain model a bounded context. A bounded context includes the code artifacts that implement the model. When using the microservice architecture, each bounded context is a service or possibly a set of services. We can create a microservice architecture by applying DDD and defining a service for each subdomain.



NOTE: Using DDD Sub-Domain approach to decompose a domain ensure avoidance of the god class.

Decomposition Guidelines

- *Single Responsibility Model (SRP)- A class should have only one reason to change.* Robert C. Martin
- *Common Closure Principle (CCP) - The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.* Robert C. Martin

Obstacles in decomposing services:

1. Network Latency
2. Self-Contained Services
3. Challenges in maintaining data consistency across service.
4. God Class – Single point of failure (God Class def: *One Class to rule them all, and in the darkness bind them.)*

Chapter 3: Inter-process communication in microservice architecture

Inter process communication - Process that helps communicate between the processes/components using wide varieties of styles & interaction patterns are called as inter process communication.

The Primary types of inter process communication – one-to-one and one-to-many each of these types are further divided into Synchronous and Asynchronous categories.

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

- ***Request/response***—A service client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. It might event block while waiting. This is an interaction style that generally results in services being tightly coupled.
- ***Asynchronous request/response***—A service client sends a request to a service, which replies asynchronously. The client does not block while waiting, because the service might not send the response for a long time.
- ***One-way notifications***—A service client sends a request to a service, but no reply is expected or sent.
  The following are the different ways of the one-way communication.
  - Publish/subscribe—A client publishes a notification message, which is consumed by zero or more interested services.
  - Publish/async responses—A client publishes a request message and then waits for a certain amount of time for responses from interested services.

API first approach – Define your API interfaces shared & get reviewed with the client teams. Over the time as the api evolved create new version.

Versioning Strategy - USE SEMANTIC VERSIONING: Semantic version or Semvers in short required the version number to be consist of three parts – **MAJOR**.**MINOR**.**PATCH**. e.g., 3.1.12. Any changes that are backward compatible are represented as incremental MINOR version number, breaking changes - changes that are NOT backword compatible are represented by incremental MAJOR changes.

Messaging Format – There are two primary messaging format TEXT base (jSON , XML) and BINARY base (Protocol/ Avro). Important difference between protocol buffer and Avro message formats - Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages.

Synchronous Messaging

REST
All HTTP based APIs are not necessarily are RESTFUL API, REST maturity model. Leonard Richardson defines a very useful maturity model for REST that consists of the following levels.

(http://martinfowler.com/articles/richardsonMaturityModel.html)

- ***Level 0***—Clients of a level 0 service invoke the service by making HTTP POST requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (for example, the business object), and any parameters.
- ***Level 1***—A level 1 service supports the idea of resources. To perform an action on a resource, a client makes a POST request that specifies the action to perform and any parameters.
- ***Level 2***—A level 2 service uses HTTP verbs to perform actions: GET to retrieve, POST to create, and PUT to update. The request query parameters and body, if any, specify the actions' parameters. This enables services to use web infrastructure such as caching for GET requests.
- ***Level 3***—The design of a level 3 service is based on HATEOAS (Hypertext As The Engine Of Application State) principle. The basic idea is that the representation of a resource returned by a GET request
  contains links for performing actions on that resource. For example, a client can cancel an order using a link in the representation returned by the GET request that retrieved the order. The benefits of HATEOAS include no longer having to hard-wire URLs into client code

Advantage and Drawback of using REST.

Advantages

- It is simple and familiar.
- One can test an HTTP API from within a browser using, for example, the Postman
- plugin, or from the command line using curl (assuming JSON or some other text format is used).
- It directly supports request/response style communication.
- HTTP/REST is firewall friendly.
- Doesn't require an intermediate broker, which simplifies the system's architecture.

Disadvantages

- Only Supports request/response model, does not support async request/response model.
- Reduce overall availability of the system – if one of the services is unavailable, the entire system is unavailable.
- Client must know the service location or service discovery service location.
- Fetching multiple resources can be challenging.
- In some scenarios – update actions are difficult to map to HTTP Verbs.

Even with the above disadvantages the REST is the defector protocol for developing microservices.

gRPC

gRPC is a binary message-based protocol. gRPC APIs using a Protocol Buffers-based IDL, which is Google's language-neutral mechanism for serializing structured data. You use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages, including Java, C#,NodeJS, and GoLang.

gRPC uses Protocol Buffers as the message format. Protocol Buffers is an efficient, compact, binary format. It's a tagged format. Each field of a Protocol Buffers message is numbered and has a type code. A message recipient can extract the fields that it needs and skip over the fields that it doesn't recognize. As a result, gRPC enables APIs to evolve while remaining backward-compatible.

A gRPC API consists of one or more services and request/response message definitions. A service definition is analogous to a Java interface and is a collection of strongly typed methods. As well as

supporting simple request/response RPC, gRPC support streaming RPC. A server can reply with a stream of messages to the client. Alternatively, a client can send a stream of messages to the server.

Advantages of gRPC

- It is straightforward to design an API that has a rich set of update operations.
- It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- Bidirectional streaming enables both RPI and messaging styles of communication.
- It enables interoperability between clients and services written in a wide range of languages.

Disadvantage of gRPC

- It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.
- Older firewalls might not support HTTP/2.
- Like REST it's a synchronous communication mechanism.
- There is a risk of partial failure.

Partial failure / Cascade failure

When one service calls another service and that calls another service, a single(partial) failure can lead to a cascade failure. The ways to prevent partial failure to lead to a cascade failure.

- Design robust Proxy Service that can handle remote service failures.
- Design mechanism to recover from a failure.

Ways to implement robust proxy

- Always use timeout – never wait for a response indefinitely.
- Limit the outstanding requests from client to a service.
- Introduce Circuit Breaker pattern.

Ways to implement mechanism to recover from a failure.

- Implement circuit breaker
- Introduce API gateway – when remote service is not responding return cached response for error response.
- Use Service Discovery – use service discovery to get to service instance endpoint. Have client-side load-balanced (client lookup service discovery, identifies different service instance endpoints and chose one of the service instance endpoints to send its request.) || application side service discovery has one major drawbacks – one need to create service discovery to be available in all possible language, this drawback can be overcome by implementing platform side service discovery where each of the service is given a DNS name /VIP – when a request is routed to a particular DNS name/VIP, the platform forward the request to one of instances of the service register registered under the DNS name/VIP. The benefit of using platform service discovery is service registration, service discovery, service health check, request routing everything is managed by the deployment platform itself. Only limitation of the platform service discovery is, it only helps discover services that are deployed in its own platform.

Asynchronous Messaging

Message – Messages has a header and a body**.** Different types of messages are

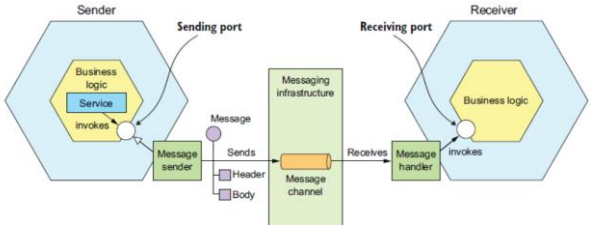
- **Document**: generic message that contains only data, consumer decides how it want to act on the message.
- **Command**: similar to RCP contains the operation & and the parameters to invoke the operations.

- **Event**: message that represent a notable change in the sender system, consumer can act on it or chose to ignore.

Message Channel – There are two type of messaging channels :
- Point-to-Point (for one-to-one communication)
- Pub-Sub (for one-to-many communication)

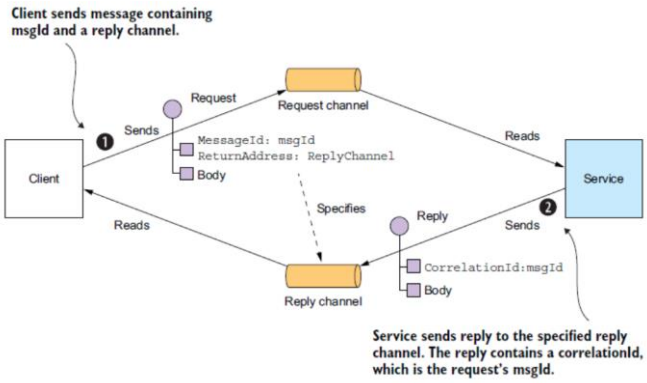
How messaging channel works?

The business logic in the sender invokes a sending port interface, which encapsulates the underlying communication mechanism. The sending port is implemented by a message sender adapter class, which sends a message to a receiver via a message channel. A message channel is an abstraction of the messaging infrastructure. A message handler adapter class in the receiver is invoked to handle the message. It invokes a receiving port interface implemented by the consumer's business logic. Any number of senders can send messages to a channel. Similarly, any number of receivers can receive messages from a channel.



Asynchronous messaging

Message will have a message id embedded in the message, which sender will send back as correlation id.



Pub/Sub Domain event.
Services use publish/subscribe to publish domain events, which represent changes to domain objects. The service that publishes the domain events owns a publish-subscribe channel, whose name is derived from the
domain class.

NOTE: it's best practice to describe the message format using JSON, XML or protobuf format, similarly for REST endpoints Open API standards.

Messaging Brokerless Architecture / Messaging Broker Architecture

Messaging brokerless architecture advantages:

- Lighter traffic, improve latency – as messages doesn't need to travel through a broker layer, it has less traffic overhead and better latency.
- Eliminate any possibilities of having performance bottleneck at the messaging broker layer.
- Less operational overhead as there is no messaging broker layer to maintain.

Messaging brokerless architecture disadvantages:

- Services communicating among themselves should know each other location Or should have a discovery mechanism to discover message locations.
- Reduce overall availability of the system as both sender and receiver should be available.
- Having QoS of guaranteed delivery is difficult to achieved.

Messaging brokerless architecture base tool - ZeroMQ (http://zeromq.org)

Messaging Broker Architecture – *as compare to brokerless architecture there are tools/platforms that support messaging broker-based architectures. Tool like ActiveMQ, RabbitMQ, Apache Kafka , AWS Kinesis , Google pubSub , Azure event-grid* **.**

While deciding on the messaging platform one needs to keep in mind the following points:

- Supporting Programming languages.
- Supporting Messaging Standards
- Messaging Ordering
- Delivery Guarantees
- Message persistence
- Durability
- Scalability
- Latency
- Competing consumer – When there are more than one instance of the receiver, there is a chance of getting the message process multiple times, messaging platform should have a mechanism to prevent such scenario to happen. Another scenario – when a receiver is still processing createOrder event, another receiver already processed cancelOrder. To prevent such scenarios – message broker implements shards

All messaging platform (broker) has their own trade-off, one needs to evaluate their requirement and select a messaging platform.

Message broker	Point-to-point channel	Publish-subscribe channel
JMS	Queue	Topic
Apache Kafka	Topic	Topic
AMQP-based brokers, such as RabbitMQ	Exchange + Queue	Fanout exchange and a queue per consumer
AWS Kinesis	Stream	Stream
AWS SQS	Queue	—

Messaging broker architecture advantages:

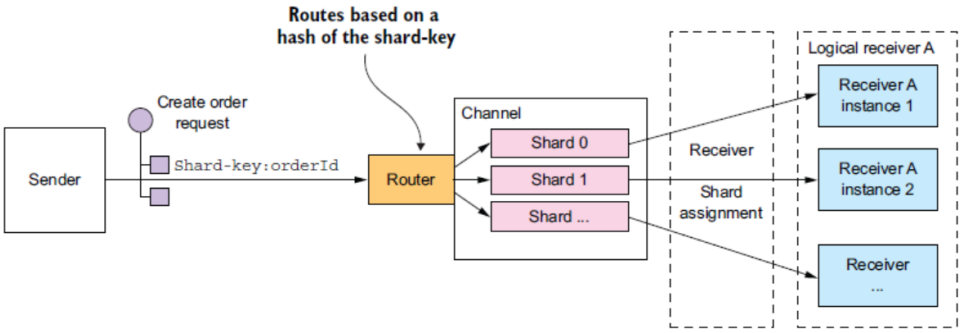
- Loosely coupling
- Message buffer
- Flexible communication – messaging broker support all type of communication, point-to-point, pub-sub, one-way.
- Explicit inter process communication – a messaging broke allow sender to communicate to its remote or a local receiver in the same way, making the process standard for inter process communication.

Messaging broker architecture disadvantages:

- Potential Performance bottleneck.

- Potential Single-point-of failure.
- Operation complexity.

Shading and Completing customers/ How to maintain message ordering?



- Messaging broker have channels, channels are divided into shards.
- Message publisher send shard key (arbitrary string) in the message header.
- Message broker based on the shard key assign message to one of the shards within the message channel.
- Message consumers are grouped together and assigned a particular shard – message arriving in that shard are send to the registered consumer of that shard. Message consumer manage the mapping of the shard and the consumers – based on registered consumer availability.

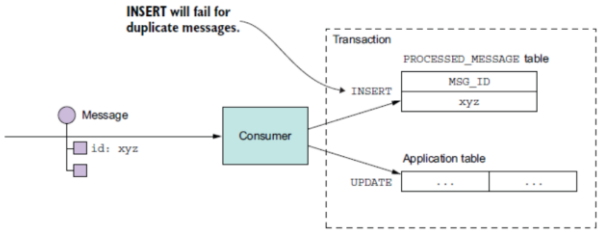
For example, Order ID can be arbitrary string value assigned as shard key – this will ensure order events related to a single orderID will always be processed by the same consumer group in the order they have been send to the message channel.

How to handle duplicate messages?

Maintaining QoS as *exactly once* can be very expensive, so most of the messaging broker ensure QoS as *atleast once*, which increases the possibility of delivering duplicate messages. In order to prevent duplicate messages, one needs to *a) write idempotent message handler b) track message been procced and reject duplicate messages.*

idempotent message handler ensures, even if duplicate message been processed the business logic is idempotent and does not deviate the logic if processed multiple times. Mostly, creating idempotent application logic is difficult, so one has to tract messages and reject duplicate messages.

One can create a transaction table with message ID a primary key, when duplicate message arrives it throws primary key constrain violation while updating the transactional table there by avoiding duplicate update on the application table(s). Alternatively, one can consider storing the messageID on the application table which will avoid the need to maintaining a transactional table, this approach is ideal for noSQL application db.

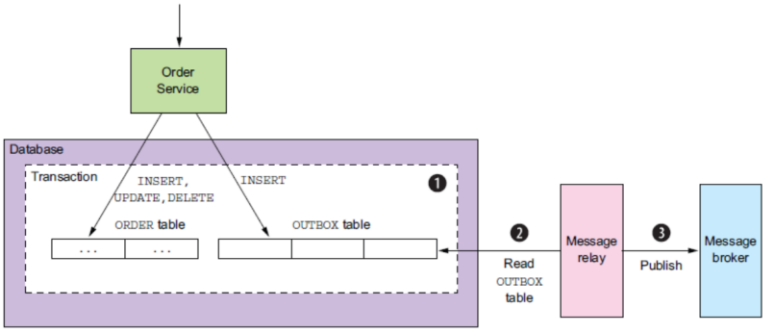


Transactional Messaging

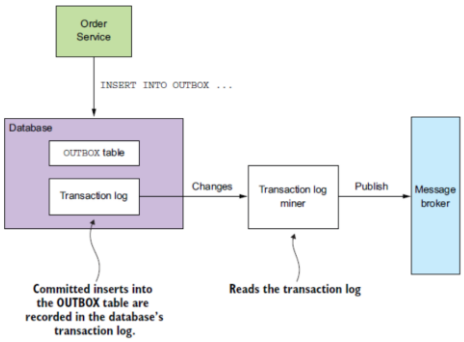
A service often needs to publish messages as part of a transaction that updates the database. Many modern message brokers including kafka don't support distributed transactions.

Alternative available for transactional messaging:

Use a transactional (OUTBOX) table, have a dbpoller messaging relay to poll the outbox table and send the message to the message broker.



Using transactional log tailing pattern : Have a transaction logs been monitored by a transaction log miner and have it send to messaging broker.



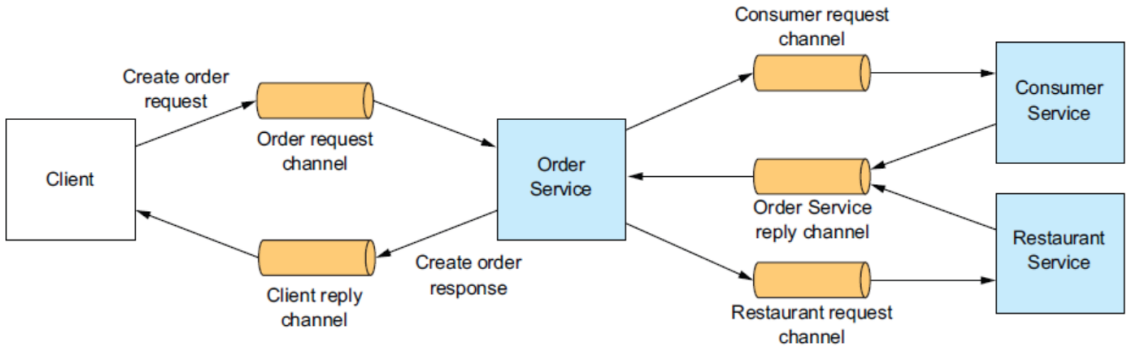
There are a few examples of this approach in use:

- ***Debezium*** (http://debezium.io)—An open-source project that publishes database changes to the Apache Kafka message broker.
- ***LinkedIn Databus*** (https://github.com/linkedin/databus)—An open source project that mines the Oracle transaction log and publishes the changes as events. LinkedIn uses Databus to synchronize various derived data stores with the system of record.
- ***DynamoDB streams*** (http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html)—DynamoDB streams contain the time-ordered sequence of changes (creates, updates, and deletes) made to the items in a DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.
- ***Eventuate Tram*** (https://github.com/eventuate-tram/eventuate-tram-core) open source transaction messaging library that uses MySQL binlog protocol, Postgres WAL, or polling to read changes made to an OUTBOX table and publish them to Apache Kafka.

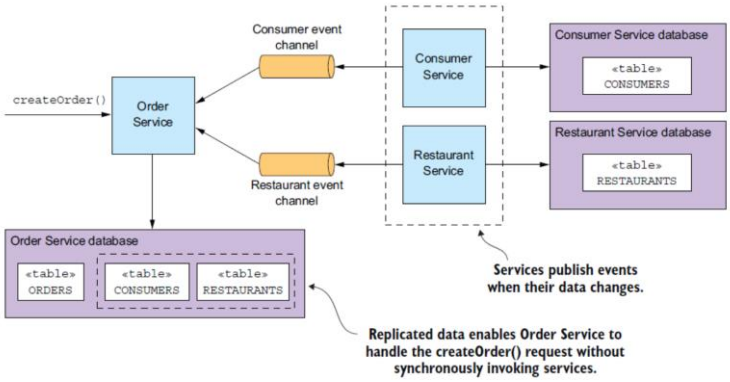
To maximized the availability of the system, one needs to minimized the synchronous communication.

How to minimized the synchronous communication?

- Use Asynchronous communication style where possible. Replace synchronous communication with asynchronous communication.



- Replicate data, maintain an updated copy of the replicated data locally and process request based on the locally available data only.

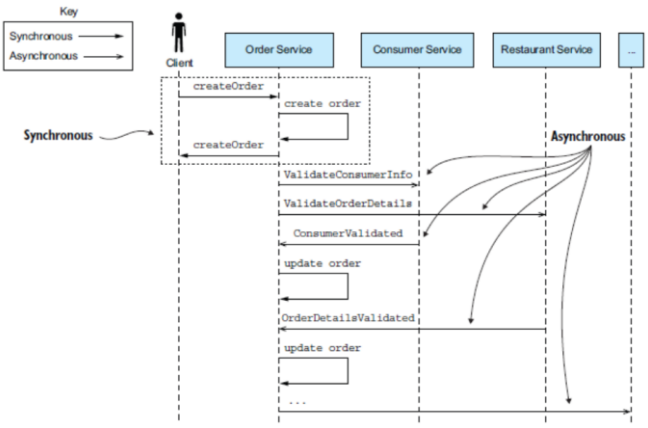


- Finish processing after returning a response: Another way to eliminate synchronous communication during request processing is for a service to handle a request as follows:
  1. Validate the request using only the data available locally.
  2. Update its database, including inserting messages into the OUTBOX table.
  3. Return a response to its client.

  A drawback of a service responding before fully processing a request is that it makes the client more complex.

For example:

1. Order Service creates an Order in a **PENDING** state.
2. Order Service returns a response to its client containing the order ID.
3. Order Service sends a ValidateConsumerInfo message to Consumer Service.
4. Order Service sends a ValidateOrderDetails message to Restaurant Service.
5. Consumer Service receives a ValidateConsumerInfo message, verifies the consumer can place an order, and sends a ConsumerValidated message to Order Service.
6. Restaurant Service receives a ValidateOrderDetails message, verifies the menu item are valid and that the restaurant can deliver to the order's delivery address, and sends an OrderDetailsValidated message to Order Service.
7. Order Service receives ConsumerValidated and OrderDetailsValidated and changes the state of the order to **VALIDATED**.



NOTE: ***Statically Typed Vs Dynamically Typed Languages*** *In statically typed programming languages, type checking occurs at compile time. At compile time, source code in a specific programming language is converted to a machine-readable format. This means that before source code is compiled, the type associated with each and every single variable must be known. Some common examples of programming languages that belong to this category are Java, Haskell, C, C++, C#, Scala, Kotlin, Fortran, Go, Pascal, and Swift. Conversely, in dynamically typed languages, type checking takes place at runtime or execution time. This means that variables are checked against types only when the program is executing. Some examples of programming languages that belong to this category are Python, JavaScript, Lisp, PHP, Ruby, Perl, Lua, and Tcl.*

When an interface changes in statically typed Language – it fails in all the integration points during compile-time where update is necessary, unlike in dynamic typed languages it wouldn't be identified until runtime.

"*Be conservative in what you do, be liberal in what you accept from others.*" *- Robustness principle*

Chapter 4: Managing transaction with Saga

Transaction is important to any important enterprise applications, in distributed application is difficult to have regular transactions, thus it advisable to have saga transactions (message driven sequence of local transactions). In case of saga, due to lack of isolation feature of ACID transactions its compensated by *countermeasures*, which prevent/reduce the impact of anomalies due to lack of isolation.

There are two possible ways to implement saga transactions – one using ***saga: choreography*** where participants exchange messages without any centralized point-of-control and the other is using ***saga: orchestration*** where centralized control tells the saga participants the operations they need to perform.

In monolithic application the transactions management is straightforward unless it involves multiple databases, message brokers and remote services.

In case of Micorservice (distributed system) there are multiple service with each of the service having their own databases. AS CAP theorem suggest only two of the three things (consistency, availability, partition) can be achieved – modern (distributed) application needs availability over consistency.

saga: choreography

In case of saga transitions, it's a series of small independent transactions within each of the systems. Its mandate to have systems communicated through message broker, because when a transaction fails the successful transactions completed prior to the failed transactions needs to be compensated unlike in case of monolithic application where XA transactions are used which can be easily rollback on an event of any failure as part of two-phase commit. Message brokers comes handy as it buffers the messages in case the system is unavailable to carry out compensation transactions.

Design consideration for saga: choreography – it needs to ensure that database update and event publish happens as part of the same transactions. Only for update/create actions there need to be an equivalent compensation event for read operation there is no need for a compensation activity.

Advantage of Saga: Choreography

- **Simple to implement**, service participating on a saga chorography transactions, need to publish events corresponding to their operation update event for update operation, delete event for delete operation and create event for create operation.
- **loosely coupled nature**, allow participant to participant in a saga chorography transaction without any need to know the other participants participating in the same transactions.

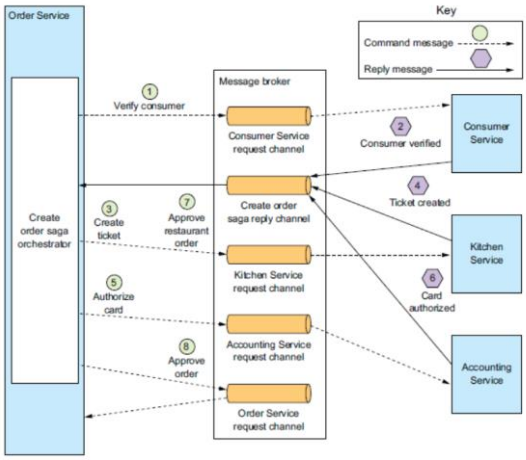
Disadvantages of Saga: Choreography

- Difficult to follow a transaction, as the components are loosely coupled sometime getting the complete picture is difficult.
- Changes of having cyclic dependencies, since their lack of clarity of the big picture there are chances of having cyclic dependency. Order Services → Account Service → Order Services → never ending loop.

- Risk of tight coupling.

Note: Due to its simplicity in nature, Saga: Choreography are well suited for the simple transactions but due to lack of visibility it's not advisable for complex operation. For more complex transactions it's advisable to use Saga: Orchestration.

Saga: Orchestration.
In case of the saga orchestration, the orchestration class is the sole responsibility to tell the participant classes which operation to execute. The below image illustrates how Saga orchestrator works.



Saga orchestrates are developed over a state machine, at every state the state is persisted and next activity is called.

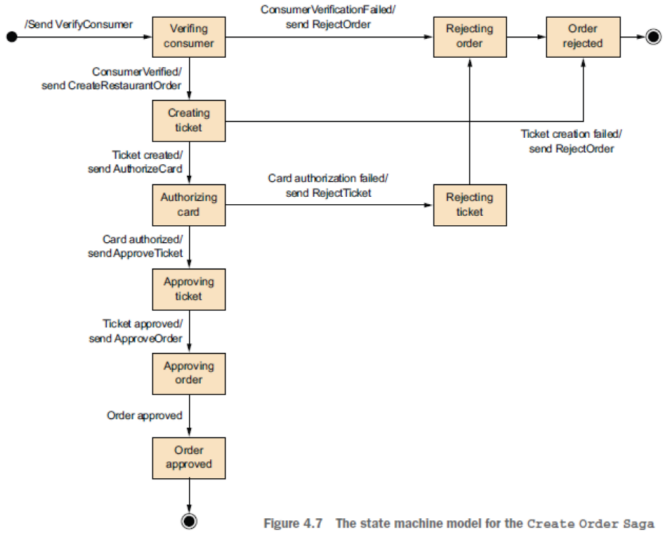


Figure 4.7 The state machine model for the `Create Order Saga`

Advantages of saga orchestration.

- **Simplified dependencies** – due to the presence of the orchestration class, the dependencies are much simplified and chance of having cyclic dependency largely reduces.
- **Loosely couple** – still the participating component service need not to be aware of the other services that are involved on the same saga orchestration process.
- **Improve separation of the concerns and improves business logic** - due to the presence of a saga orchestration class, the saga transaction logic can be localized to the orchestrator component – the saga participant can have clear business logic. The domain object can be

made simpler as it does not need to be aware of the saga transaction which its going to be a part of.

Disadvantage of the saga orchestration

- As orchestrator class is the central component there is a chance of having all too much of business logic in the orchestrator as result it becomes smart orchestrator tell dumb services what they need to do. This scenario can be avoided by ensuring saga orchestrator having only sequencing logic and no business logic at all.

Another problem with the distributed transactions is the lack of isolation.

How to handle lack of isolation.

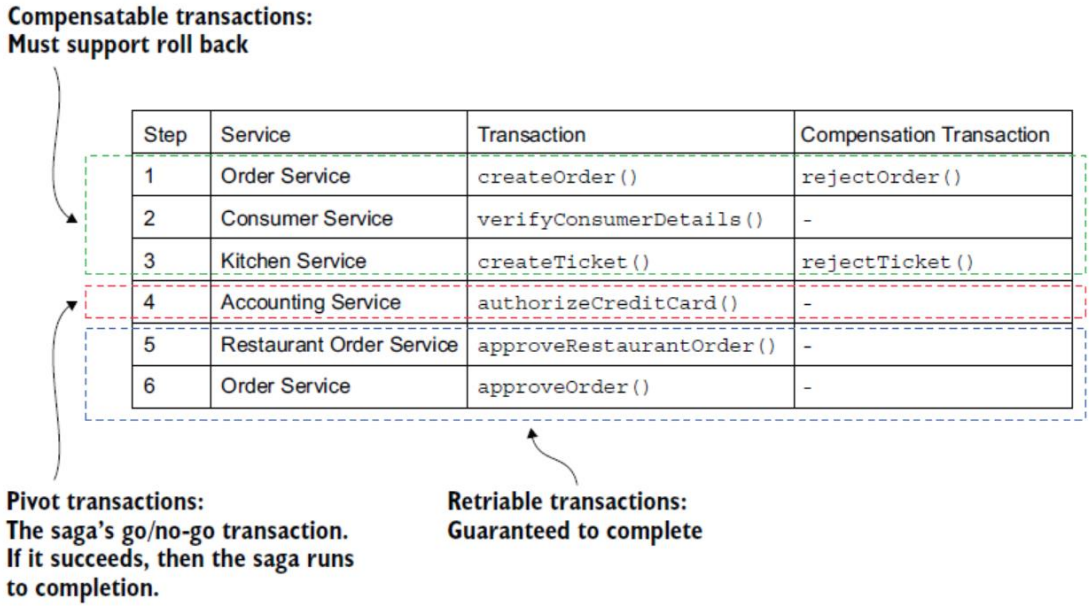
Isolation is very important. It ensures that the transactions do not impacts by the read and write operation of the separate transactions. In case of distributed transactions, it's hard to achieve as in saga transactions, transactions are committed to the database while saga transaction are still in-progress, other services can read the committed while the transaction is not fully completed.

There are possibly three major issues which can be arises due to these anomalies.

- Lost update – saga transactions override the data without reading the data updated by other services.
- Dirty read – saga transactions reading those updates that are yet to be completed.
- Fuzzy/nonrepeatable read – same saga process reading same data from two separate places and getting different results.

Saga transactions types

- **Compensation transactions**: those transactions that needs a rollback transaction to compensate the failed transactions.
- **Pivot transactions**: go/no-go type of transactions, if the transaction is successful the saga will run until completion
- **Retriable transactions**: transactions that are followed by Pivot transactions – they can be retried until success.

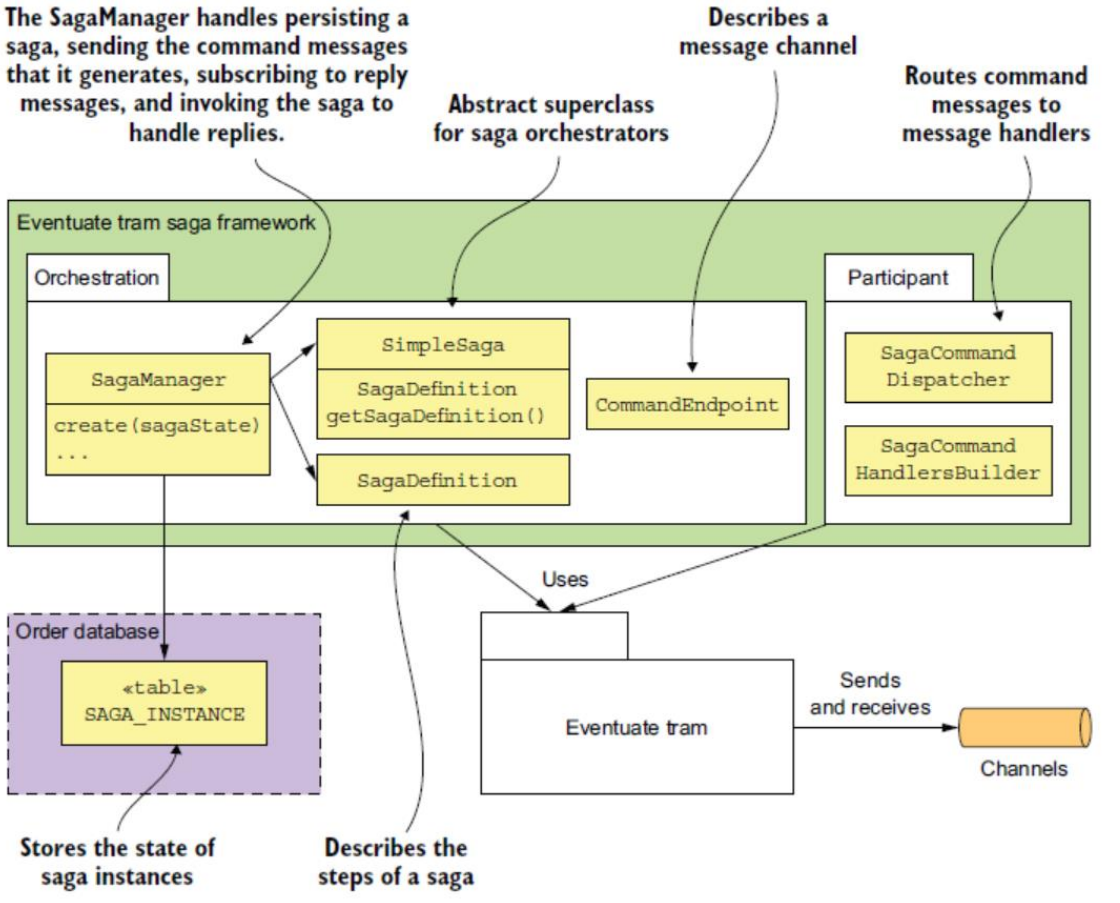


Step	Service	Transaction	Compensation Transaction
1	Order Service	`createOrder()`	`rejectOrder()`
2	Consumer Service	`verifyConsumerDetails()`	-
3	Kitchen Service	`createTicket()`	`rejectTicket()`
4	Accounting Service	`authorizeCreditCard()`	-
5	Restaurant Order Service	`approveRestaurantOrder()`	-
6	Order Service	`approveOrder()`	-

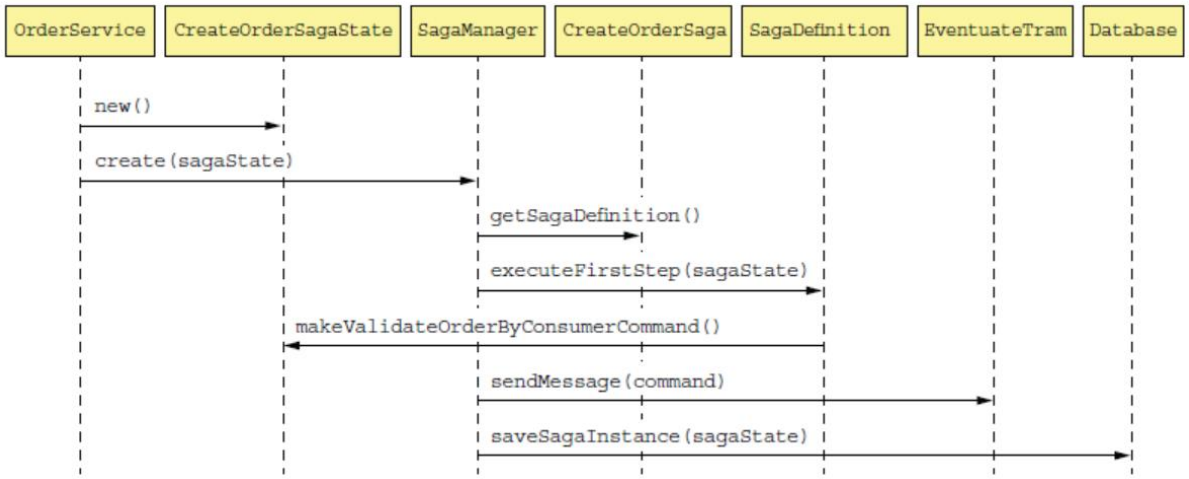
Countermeasures to handle lack of isolation

- Sematic lock – An application-level lock. For compensation transaction to update a field that can define that sga transaction is NOT COMPLETED thereby locking other transaction to use the data. In case of Order saga, order.status field can be updated with *_PENDING status which can act a flag for other transactions stating that the saga transaction is NOT completed.
- Commutative Update – Design update operation to be executable in any order.
- Pessimistic view – Reorder the steps within saga to minimized the business risk. For example, in the create order scenario – where due to a dirty read of the available credit limit value an order is created that exceeded the customer credit limit to minimized the risk reorder the order cancellation saga to with retriable transaction to increase the customer credit limit.
    1. OrderService - Cancel the order – update the order status to Cancelled.
    2. DeliveryService - Cancel the delivery service.
    3. CustomerService – increase the customer limit.
- Reread values – Prevent dirty ready by rereading the data to verify the data is unchanged before updating.
- Version file – Record the update to a record so that they can be reordered. In case of the cancellation saga – when an order is cancelled the AccountService – Cancel Authorization event is sent before AccountService – Card Authorization. AccountService reorder the request before executing it as it has already received Cancel Authorization it will not perform Card Authorization request.
- BY value – Use each request business risk to dynamically select the concurrency mechanism.

EVENTUATE TRAM SAGA FRAMEWORK OVERVIEW



```
public CreateOrderSaga(OrderServiceProxy orderService, ConsumerServiceProxy consumerService, KitchenServiceProxy kitchenService,
                       AccountingServiceProxy accountingService) {
  this.sagaDefinition =
          step()
           .withCompensation(orderService.reject, CreateOrderSagaState::makeRejectOrderCommand)
          .step()
           .invokeParticipant(consumerService.validateOrder, CreateOrderSagaState::makeValidateOrderByConsumerCommand)
          .step()
           .invokeParticipant(kitchenService.create, CreateOrderSagaState::makeCreateTicketCommand)
           .onReply(CreateTicketReply.class, CreateOrderSagaState::handleCreateTicketReply)
           .withCompensation(kitchenService.cancel, CreateOrderSagaState::makeCancelCreateTicketCommand)
          .step()
              .invokeParticipant(accountingService.authorize, CreateOrderSagaState::makeAuthorizeCommand)
          .step()
           .invokeParticipant(kitchenService.confirmCreate, CreateOrderSagaState::makeConfirmCreateTicketCommand)
          .step()
           .invokeParticipant(orderService.approve, CreateOrderSagaState::makeApproveOrderCommand)
          .build();
```



The sequence of events is as follows:

1. Eventuate Tram invokes SagaManager with the reply from Consumer Service.
2. SagaManager retrieves the saga instance from the database.
3. SagaManager executes the next step of the saga definition.
4. CreateOrderSagaState is invoked to generate a command message.
5. SagaManager sends the command message to the specified saga participant (Kitchen Service).
6. SagaManager saves the update saga instance in the database.

If a saga participant fails, SagaManager executes the compensating transactions in reverse order.

What is mean by isolation in ACID transactions?

Isolated transactions are considered to be “serializable”, meaning each transaction happens in a distinct order without any transactions occurring in tandem. Any reads or writes performed on the database will not be impacted by other reads and writes of separate transactions occurring on the same database. A global order is created with each transaction queueing up in line to ensure that the transactions complete in their entirety before another one begins.

Chapter 5: Designing Business logic in microservice architecture

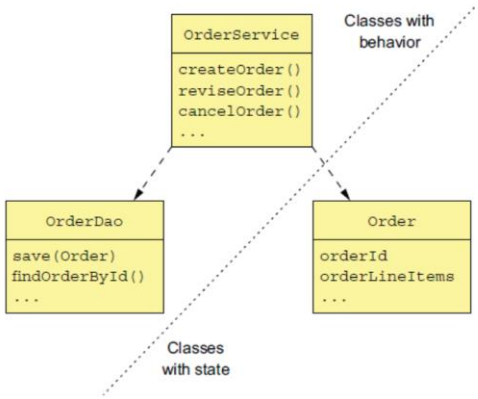
Challenges in designing business logic in microservice architecture

1. Need to define business logic well within the service boundaries.
2. Need to define business logic that works within the transaction management constrains of the microservice architecture.

The above two challenges can be addressed using the aggregator pattern from Domain Driven Design. Aggregator is a cluster of objects treated as a single unit.

- Aggregator avoids having object reference spanning across service boundaries, all inter-aggregator reference is referenced by primary key instead of object reference.
- Because the transactions can only create or update an aggregator object – it works fine with the microservice transactional model.

There are two possible approaches for developing a business logic – Procedural Transaction Script Pattern AND Object-Oriented Domain model pattern.



If the business logic is simple, it's advisable to use Procedural Transaction Script Pattern where the business logic are part of classes that have behavioural procedures, and there are other sets of classes which contains the state for the business logic.

What is DDD?

DDD stands for Domain Driven Design, where each services have its own domain instead of having a domain model that span across the entire application. The following are key building block of domain driven design.

- **Entity**: The object that has a persistent identity, classes that are persisted using JPA @Entity are usually DDD entities.
- **Value Object**: A object that has collection of values, the two value objects whose attributes have same values can be interchangeable. For example, Money Class which has two attributes Currency and Value.
- **Factory**: An object or method that implements object creation logic that's too complex to be done directly by a constructor. It can also hide the concrete classes that are instantiated. A factory might be implemented as a static method of a class.

- **Repository**: An object that provides access to persistent entities and encapsulates the mechanism for accessing the database.
- **Service**: An object that implements business logic that doesn't belong in an entity or a value object.
- **Aggregates:** An aggregate is a cluster of domain objects within a boundary that can be treated as a unit. The entire aggregate object is often load/save to database.

Aggregate Rules: the following are the aggregate rule which needs to be obeyed in order to keep the aggregate self-contained unit that can enforce its invariants.

1. Reference only the aggregate roots: Any operation that needs to be performed on the aggregates entities needs to performed through the method exposed by the aggregate root.
2. Inter-aggregate reference should be using primary-key instead of object reference to ensure that aggregates boundaries are maintained and honoured. Due to the structure of the aggregates (aggregates contains multiple entries & value-objects), it's convenient to store aggregates in NoSQL.
3. One transaction creates or updates one Aggregates – unlike in monolithic application with single RDBMS where multiple transactions can be joined together with XA/2PC, in case of microservice snapping a single transaction across multiple services is not an option. Hence it is advisable to single transaction to create or update a single aggregate, each of these local transactions than can be sewed together using Saga Distributed transactions.

Aggregates granularities: As aggregates needs to be persisted/serialized it is better to keep it as small as possible, keep it small increases number of simultaneous request that application needs to handle. One need to be careful where to draw the service boundaries.

What is an event?

- Something that happens
- A noteworthy happening
- A social occasion or activity
- An adverse or damaging medical occurrence, a heart attack or other cardiac event.

In context of DDD, an event is a change of state of an entity – Order Created, Order Received, Order Shipped, Order Cancelled.

Its very import to keep other system aware of event, in order to do so event needs to be published. Here are some of the scenarios why events need to be published.

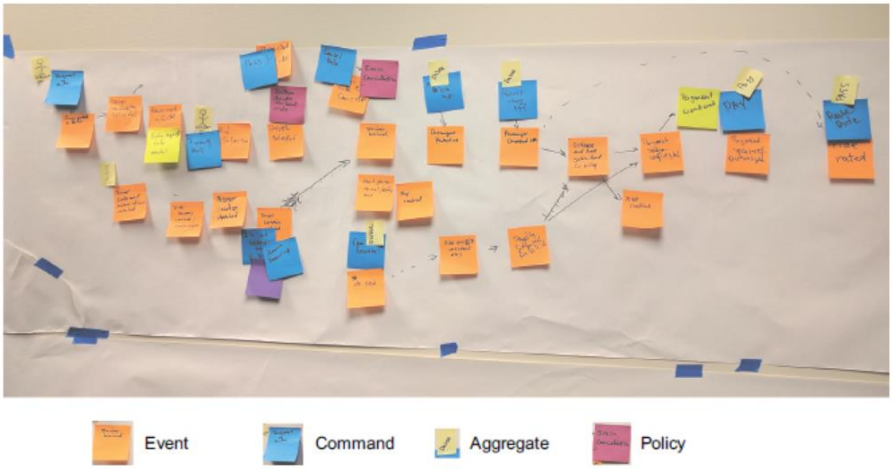
1. Maintaining consistency across services using chorography base saga.
2. For notifying the services that maintains a replica of the source data that the data has changed.
3. To notify other application via webhook or messaging broker to trigger the next step of the business processes.
4. For notifying end-user on status of their transactions
5. For monitoring domain event to ensure the application is behaving correctly.
6. For analysing events to model user behaviour.

Domain event is the mostly represented by past participle verb, it has properties that meaningfully conveys the event. Example of the domain event are OrderCreated which has property called as OrderId.

```
interface DomainEvent {}

interface OrderDomainEvent extends DomainEvent {}

class OrderCreated implements OrderDomainEvent {}

class DomainEventEnvelope<T extends DomainEvent> {
  private String aggregateType;
  private Object aggregateId;
  private T event;
  ...
}
```

The event's metadata

Event Storming, is an event-centric workshop for understanding a complex domain. The following are the key objectives which will be achieved through event storming.



1. ***BrainStrom Events:*** Domain experts will ask to laid out the domain events in a rough timeline on the surface model.
2. ***Identify Event Trigger:*** Ask the domain experts to identify the domain event trigger , trigger can be anything
    a. ***User Actions***
    b. ***External Systems***
    c. ***Another domain even***
    d. ***Passing of time***
3. ***Identify Aggregates:*** Ask the domain experts to identify the aggregates that are consumed by each command and emit the corresponding event.

NOTE: For generating event instead of using same aggregator to fetch the event details and also publish the event, it's advisable to use two separate classes – one for aggregator functions and other one for servicing (publishing) functions – that way severing function can be dependency inject into aggregator class.

Chapter 6: Developing Business logic in with Event sourcing

Though event sourcing is a lucrative and straightforward way of implementing a logic, it's also error prone. Business logic continues to executed even if the event is not/fail to publish.

Limitation of the traditional persistence storage methods in microservices contexts.

- **Object Relational impedance matching:** *fundamental conceptual mismatch between the tabular relationship schema and the graphical representation of the rich domain model.*
- **Lack of aggregation history:** *in case of traditional persistence model***,** *there is only way to store the current state of the model – current state always overrides the previous state. If an aggregated history needs to be maintained, then one needs to write complex logic which are error prone.*
- **Implementing audit logging in tedious and error prone.** *Implementing auditing logic is complex.*
- **Event publishing bolted on business on to the business logic:** *ORM frameworks don't provide any OOTB feature where events are automatically published as part of the transaction.*

Event Sourcing: *Event sourcing an event-centric technique for implementing business logic and persisting aggregates. Aggregates are store in database as a series of events, each event represents a state change of the aggregate. An aggregate business logic is structured around the requirement to publish/consume these events. Event sourcing is very different from traditional persistence storage. In case of Event Sourcing, its persistence aggregates are store as events as a series of event stored in event store. Events are re-played to recreate the state of the aggregate.*

In case of Order management solution instead of storing order information into ORDER & ORDER_LINE tables, the same info can be stored in EVENT_TABLE where each row represents a state of Order identified by a unique event_id. When a new event is updated or added, a new entry is stored under EVENT_TABLE when an aggregate needs to be loaded – all events are played for that aggregate and aggregates is created.

An event can be minimal data (just a aggregatorID) , or it can have the complete aggregator info.

How to handle concurrent update?

Order aggregate table have an additional column, called VERSION. When the aggregate is updated the VERSION value is incremented by 1. The update is only allowed when the VERSION value is unchanged from the application that reads the aggregates value. In case two concurrent update is triggered the first update will be successful, while the second update will fail as VERSION field value will change from the original value.

How to improve the performance of the Aggregate consolidation?

In order to re-create the aggregate, one needs to re-play the events. When there are huge number of events, re-play events every-time can be very costly. Hence one can persist snapshots at regular intervals. When an aggregate needs to be re-created load the snapshot and apply the events that created after the snapshot there by reducing the costly re-play actions.

Consumers should be idempotent:

Consumer Service should be made idempotent, so that an event is not processed multiple time. If the event store in implemented using a RDBMs data-source then a extra column called PROCESSED can be used to flagged event that are consumed. For nonSQL DB messageID is auto-generated and unique checking the exists of the message ensure idempotency to the consumers.

The benefit of Event Sourcing

- Reliable event publishing.
- Preserved the history of aggregates.
- Mostly avoid O/R impedance matching problem.
- Provide developer with time-machine.

The drawbacks for Event Sourcing

- **Different programming model that has learning curve:**
- **It has the complexity of the message-base application:**
- **Evolving event can be tricky:**
- **Deleting data can be tricky:** In case of event store, deleting data can be tricky, these can be address with implementing soft-delete, however GDPR (General Data Protection Regulation) grants individual right to erasure – when we cannot implement soft-delete. Another approach to implement unique encryption key for each user, when a particular user data is deleted – delete the corresponding encryption key. For aggregator keys where PII is used, replace those with UUID token, before implementing encryption key deleting, otherwise deleting aggregator key will be challenging.
- **Querying event store can be challenging:** In case of event store the event are not store in a way where it's easy to fetch a record , and if the event store in build in NoSQL it don't allow to query on non-primary fields which increase the complexity of querying the event store.

Implementing Event Store:

Event Store is a hybrid implementation of Database + Message Broker. Some of the commercially available event stores are

- EventStore : .net base open source even-store
- Lagom: microservice framework developed by LightBend
- Axon: An open-source java microservice framework developed for implementing event sourcing and CQRS
- Eventuate: Open-source framework for implementing EventSourcing – there are two version Eventuate SaaS version | Eventuate Local – with Kafka & MySQL implementation.

Event Sourcing and Saga : Event Sourcing works effectively with Saga – each of the saga participants aggregates process a command and emits another event.

Saga state should be perform atomically :

1. Saga Creation : A service must atomically create/update an aggregate & create a saga Orchestrator.
2. Saga Orchestration : Saga atomically must consume a replies, update its state and send command message.
3. Saga Participants : Saga participant must atomically consume message, detect & discard duplicates , create/update aggregates and send the reply event message.

Event Sourcing with Saga Chorography : Event Sourcing provides the required mechanism for, message-base IPC , message de-duplication , atomic update of the state and message reply, making the saga chorography implementation straightforward. However, there are few drawbacks too, event sourcing events are used for dual purpose in event sourcing saga chalcography 1. For updating the sate change and 2. Saga chorography. In an event when there is no state change it's difficult to emits events, for example : when updating an aggregate would cause business violation – in this case there is no aggregates update hence so events to raise business exceptions hence its advisable to implement complex saga's using event sourcing saga orchestration.

Event Sourcing with Saga Orchestration*: In case of the event sourcing saga orchestration – the service must perform the first action of aggregate update/create and then ensure the second action is performed eventually. This can easy archive when using RDBMs base even store – creating/updating an aggregate & creating a saga orchestrator can be done in a same ACID transaction. Achieving the same when implementing a NoSQL Event Store can be challenging as it doesn't give same ACID transactional properties as that of RDBMs event store. In order to implement Event Sourcing with Saga Orchestration using NoSQL event source – one need to implement two separate actions, create/update aggregate & have a separate event handler that can handle domain event and create the saga orchestrator.*

Domain event handler should also be able to handle duplicate event – in case there are duplicate event due to at-least once message-delivery guarantee. This can be addressed by having a unique ID sagaID, or use domain eventID as sagaID which will be unique per event. Along with this , the handler also needs to handle idempotency of the saga orchestrator, once the event is processed it should be able to identify the duplicate event and rejects it. This can be achieved by maintaining a list of processed eventIDs and rejects any event whose ID can be found in the processed list.

For sending the reply back atomically after processing there are two main challenges

1. If there is no change on aggregate event will not be generated.
2. Needs separate implementation for aggregate needs saga transaction vs aggregate don't need saga transactions.

Event sourcing saga orchestrator has three main challenges to be addressed in its implementation.

1. How to persist saga orchestrator?
2. How to atomically change saga orchestrator state and send command message?
3. How to ensure saga message reply process reply message exactly once?

Chapter 7: Implementing Quires in microservice architecture

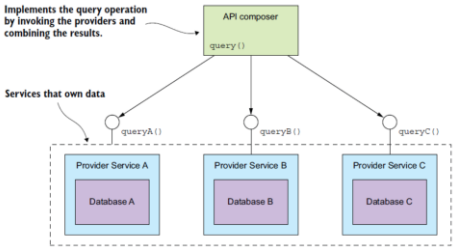
Implementing queries in microservice architecture is straightforward, as the data needs to be queried from multiple services. There are two common ways to implement quires in microservice architecture.

- **API composition pattern**: This is the simplest approach where a single service calls different client service responsible for the data and then combines the data and send it back to the caller.
- **Command Query Resource Segregation**: This is more complex than API composition pattern, where one or more view database is maintained solely for viewing purpose.

In case of monolithic application, where data resides in a single database querying aggregates that spread-across multiple services is straightforward – single join query will be able to fetch the data from multiple tables into a single response.

API composition pattern – has two components

- API Composer Service – A single service that combines all the response from the provider services. This can be done through API gateway or could be another service (frontend-for-backend) service implementation.
- Provider Services – Individual services that query specific microservice services datastore and reply back to the composer service.



Implementing API Composer pattern with synchronous HTTP/gRPC call – the provider services are behind a same firewall and can be called either in sequence/parallel then one can use this pattern. One can also use frontend for backend pattern / API gateway pattern to implement such API Composer.

Implementing API Composer pattern with reactive programming model – the provider services are need to be called in parallel without blocking all the services then one need to use reactive programming model.

The benefit of using API Composer Pattern

- Simple to implement

The drawback of using API Composer Pattern

- **Increase overhead**: Involves triggering multiple request and executing multiple queries to fetch response, this adds overall increase in the performance overhead of the system.
- **Risk of reducing availability of the overall system**: Even if one of the provider services is unavailable the entire API composer service is unavailable.
- **Lack of transaction data consistency:** Having a strong ACID consistency model across all the provider service response is extremely difficult resulting in lack in overall transaction consistency of the model.

CQRS (Command Query Resource Segregation)

This pattern is used for maintaining one-or-more database for queries in parallel to the main transactional database.

The API composer pattern is less effective in implementing query service with vast filter options, as all services don't have the required filter options. CQRS pattern can be leveraged to specifically implement query options.

It's important to store the data in easy to query database – for example, data which are query based on geospatial data , should be queried using geospatial query. CQRS pattern, helps implementing query optimized database.

Need for separating concerns: Apart from data ownership of the retrieving data, separation of concerns for a single microservices is also equally important. Its important that command and queries can be separately scalable.

The benefit of using CQRS design pattern.

- Effectively implements quires for Microservice Architecture
- Effectively implements the diverse quires.
- Make query possible to query streams event source.
- Improve on separation of concerns.

Limitation of CQRS

- Complex implementation : Managing two separates services , in some instance two separate database for improving the overall system performance which adds added complexity to the code.
- Dealing with replication lags

The choice of the database to implement CQRS can be difficult as there can be multiple database options for implementing data-store.

The following needs to consider while designing the data-store.

- Handel Concurrency: if the events are from the same aggregate, then all the events related to the aggregate will be processed sequentially, but if there are multiple aggregate sharing same view then record needs to be updated sequentially. The records need to be updated using pessimistic or optimistic locking.
- Idempotent even: Due to delivery failure single event, a single event may be resending twice/multiple times – the data source update should be idempotent even OR reject the duplicate request.

Designing for effective querying:

- Design the table for querying data.
- Design index for querying table
- Implementing query | Identify the filters
- Paginating the querying results
- Data Updation , how the data needs to be updated. Whether to use putItem() or updateItem(). putItem() create or replace the entire item matching the primary key , while updateItem() updates or creates the individual attribute of the item.
- Detect duplicate items

Chapter 8: External API patterns

Monolithic application leads to Monolithic APIs. What is a monothetic API?

Different type of API clients?

- Web Application
- JavaScripts
- Mobile application
- Third Party

Difficulties in leveraging the approach used in microservice:

- Fine grain service needs clients to make multiple invocation.
- Lack of encapsulation makes API changes difficult, *Exposing the finer service to the client can be a problem, as the service changes the client also needs to make the change at their end.*
- Services expose through IPC (*AQMP or gRPC*) are difficult to use outside a firewall.

API Gateway Pattern

An API gateway is a service that's the entry point into the application from the outside world. It's responsible for request routing, API composition, and other functions, such as authentication.

Responsibilities of API Gateway

- Request Routing
- API Composition
- Edge Functions
- Protocol transaction

Benefits of API Gateway

- API Composition
- Protocol Translation
- Providing each client with client specific APIs
- Implementing Edge functions like
    - Authentication
    - Authorization
    - Rate Limiting
    - Caching
    - Metric Collections
    - Request Logging

PS: some of the edge functions like metric collection / caching are implemented in the backend services. This helps in reducing the network latency.

Drawback of API gateway : Single point of failure.

Challenges with API gateway

- Performance and Scalability: System I/o threads are limited, which limits the concurrent synchronous threads, as an alternative one can use non-blocking i/o threads to improve the performance and scalability of the system.
- Handling partial failures.

Ownership of the API Gateway code, its best to have API team owns the API gateway and backend services together, instead of centralized team owning all the API gateway code and been a bottle neck of the operation.

Some of the commercially available API gateways:

- AWS API Gateway : Limitation with API gateway – support only server side discoveries , also it doesn't support API composition. Supports ONLY HTTP request protocol.
- AWS Application Load Balancer : Does not full meets the API gateway retirements – authentication , API composition.
- Kong / Tarafik – open source API gateway products. Still doesn't support API composition out of the box. For other API capabilities, one can configure plugins.
- Netflix Zulu (API gateway framework) : one can create API gateway code using these frameworks.

Implementing API gateway using Graph-Based API technologies: There are two possible implementation of Graph based API technologies – graphQL and Netflix Falcor.

Netflix Falcor	**GraphQL**
Invented by Netflix.	Invented by Facebook
Implementation of a standard.	Standard by itself – with client and servers available on varieties of languages.
The Falcor client retrieves data from a Falcor server by executing a query that retrieves properties of that JSON object. The client can also update properties. In the Falcor server, the properties of the object graph are mapped to backend data sources, such as services with REST APIs. The server handles a request to set or get properties by invoking one or more backend data sources.	It models the server-side data as a graph of objects that have fields and references to other objects. The object graph is mapped to backend data sources. GraphQL clients can execute queries that retrieve data and mutations that create and update data.

#288

Backend For FrontEnd (BFF) Pattern – Created by Phil Calçado and his colleague at SoundCloud. BFF patter clearly define the responsibilities of each of the team by combining related service APIs into single service endpoint.

GraphQL framework can be leveraged to build an effective API gateway framework, where one can write graph oriented schema to describe the server side data model and its supported quires and

then map schema to the service by writing resolvers which retries data. GraphQL-based clients execute queries against the schema that specify exactly the data that the server should return. As a result, a graphQL-based API gateway can support diverse clients. At times the GraphQL queries can be ineffective as they need to go round-trip to fetch data , thus implementing a cache can overcome these limitations.

Chapter 9: Testing Microservice

Automation Testing 4 phases

- Setup – Initialized the test fixture , which consist of the SUT (subject under test) and its dependencies set to initial state.
- Exercise - invoke the STU
- Verify – make assertion about the STU outcome , state of the STU.
- Teardown – Clean-up of the test fixtures .

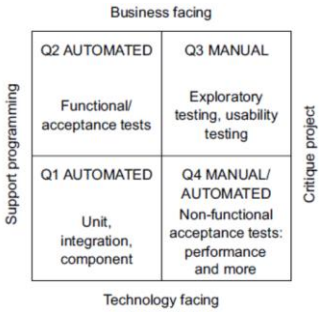
Test Stubs	Test Mocks
Test Stubs are the test double the returns values to the SUT.	A mock is a test double that test uses to verify the SUT is correctly invoking its dependencies.

Different Types of testing depending upon their scopes

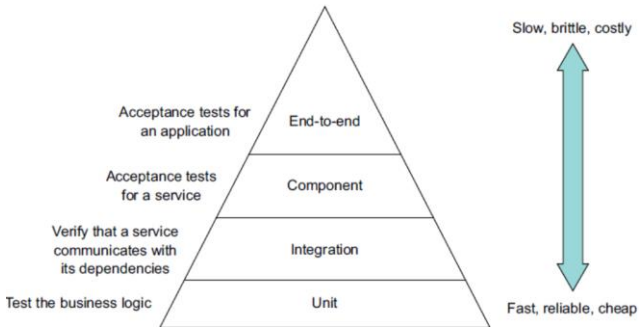
- Unit Testing
- Integration Testing
- Component Testing
- End-to-End Testing

Test Quadrant

- Q1 – Support programming / technology facing: unit testing / integration testing.
- Q1 – Support programming / business facing: component testing / end-to-end testing.
- Q1 – Critical application / business facing: Usability Testing / exploration testing
- Q1 – Critical application / technology facing: Non-functional acceptance Testing / Performance testing.



Test Pyramid



Challenges in testing microservices

- Communication with the external APIs/System
- Every evolving APIs/Services
- Communication over synchronous APIs/Services
- Consumer-Provider relationship. (Consumer Contract Testing) || Spring Cloud Contract is a consumer contract testing framework from Spring. Spring cloud contract can be used to write a contract , which can be used for generating code - consumer application can test the generated code and validate the contract before publishing it for consuming. Spring cloud contract can also be used to test messaging base interaction (contracts).

Integrating testing into Delivery pipeline - As code flows through the pipeline, the test suites subject it to increasingly more thorough testing in environments that are more production like. At the same time, the execution time of each test suite typically grows. The idea is to provide feedback about test failures as rapidly as possible.

Testing Strategy

- **Entities** (object with persistence identity) are testing using sociable unit tests.
- **Value Objects** (objects that are collection of values) are tested using sociable unit tests.
- **Saga** (classes for maintain data consistency across services) are tested using sociable unit tests.
- **Domain Services** (classes for implementing business logic) are tested using sociable unit tests.
- **Controllers** (classes handling HTTP Requests) are tested using sociable unit tests.
- **Inbound / outbound messaging gateway** are tested using sociable unit tests.

Chapter 10: Testing Microservice 2

Integration test – where infrastructure services are integrated to validate the business logic in isolation.

Integration Testing Strategy, testing the service adapters or testing the service contracts (a contract is a concert example of services between a pair of services.)

- Mocking Adapters

- Testing contracts – there are two possible options of testing a contract, *consumer side testing* where provider is simulated by stubs & *provider side testing* where consumer adapter is mocked.

NOTE: in case of the unit testing the data is not persistence, but in case of the integration testing the data needs to be persisted in datastore.

Component testing – are user acceptability tests for the service.

Component test verify the behaviour of the in isolation, it might be in in-memory version of the infrastructure such as database, making it easier to write. Component testing are mostly written in Gherkin language (testing DSL).

For example, for the given story,

```
As a consumer of the order service, I should be able to place and order.
```

The above scenario can be break down into following acceptance criteria, written in Gherkin language structure where Given is the pre-conditions, when is the action or the event occurred and then/and are the expected outcome.

```
Given a valid customer.

Given using a valid credit card.

Given the restaurant is accepting order.

When I place an order for chicken vindaloo at Ajanta

Then the order should be APPROVED

And an OrderAuthorization Event should be published.
```

The above acceptance criteria can be tested using.

1. `Create an order invoking POST/Orders endpoint`
2. `Verify the state (status) of the order by invoking POST/Orders/{orderID} endpoint.`
3. `Verify the Order Service publish OrderAuthorization Event by stubbing appropriate messaging channel.`

There are two ways to test component testing.

- In-process testing
- Out-of-process testing

In case of the in-process component testing, the dependencies are stub / mocked and then using springbootTest framework run the test in same JVM.

In case of the out-of-place component testing approach, the service under test are package in production ready format and tested as a separate process making it more realistic – at the same time its slower and also brittle. Stubbing out-of-place component testing approach , one can use spring cloud contract framework to stub the service endpoints , the drawback of this approach is since spring cloud contract focuses on contract validation it bit heavyweight. Another approach is to HTTP stubs using wire mock.

End-to-End testing – testing group of services together, this is the top most level of the testing pyramid, and should be used sparingly.

Writing end-to-end test is expensive and slow in execution, as it needs all the components to be deployed in their respective containers and then test the flow. Its high likely that few of the component may have issues with the deployment leading to a failed test. It's advisable to minimize the end-to-end testing as possible.

Chapter 11: Deploying Production Ready Services

Three main challenges for developing a production ready application.

1. **Application Security:** Application should be safe, there should not be any known security vulnerability.
2. **Service Configurability**: All application configuration should be externalized; they should not be hard wire to the code.
3. **Observability**: To understand and troubleshoot the application, each request should be traceable within the application.

Application Security: Application security is foremost important aspect of production ready code. An application developer is responsible for

a. **Authorization**: Verify the identity of the principal (a principal can be human or a system to which identity is attached to).
b. **Authentication**: Verify if principal has authorization (access) to perform the requested operation. An operation on an object or on an aggregate can be restricted through Access Control List (ACL) which give certain grants certain privileges to a ROLE(S). This is also known as ROLE BASE ACCESS CONTROL (RBAC).
c. **Auditing**: Tracking operation each principal performs on any operation.
d. **Secure Inter-process communication**: all internal communication should be over Transport Security Layer (TLS) and should have mechanism to authentication the requester.

Implementing Security in Monolithic applications.

Most monolithic application build in JAVA, use Spring Security Framework where once the user logs in into application and successfully authenticate, JSESSIONID session token is returned back to the user, user then include the same session id for all its requests. Framework interceptor will then incept the request, authenticate the session token and update security context with principal (user) ROLE details.

Popular JAVA / NodeJs Security frameworks are

1. Spring Security (JAVA) security framework
2. Apache Shiro (JAVA) security framework
3. Passport (NodeJS) security framework

In case of microservices, monolithic security mechanism cannot be implemented as is for following two reasons.

1. No in-Memory security context been share by all microservices.
2. No centralized session context available.

When it comes to microservice, authorization and authentication needs to be handled separately.

Handling authentication at API Gateway level.

Handling authentication at each service level can pose serious consequence if any of the services fails to implement authentication or implements weak authentication the entire system will be at risk, also complexity of the authentication needs to be repeated at each service level.

Better option of implementing authentication logic at the API Gateway level, that way we can ensure same level of authentication been performed for all services, also the complexity of implementing different authentication mechanism can be encapsulated. Once the API gateway authenticates a request, it would send back a token to the client, client then needs to add the same token for its subsequent requests.

Handling authorization for microservices

There token send back to the client after successful authentication be used for implementing authentication at service level. However, there are two types of tokens usually used

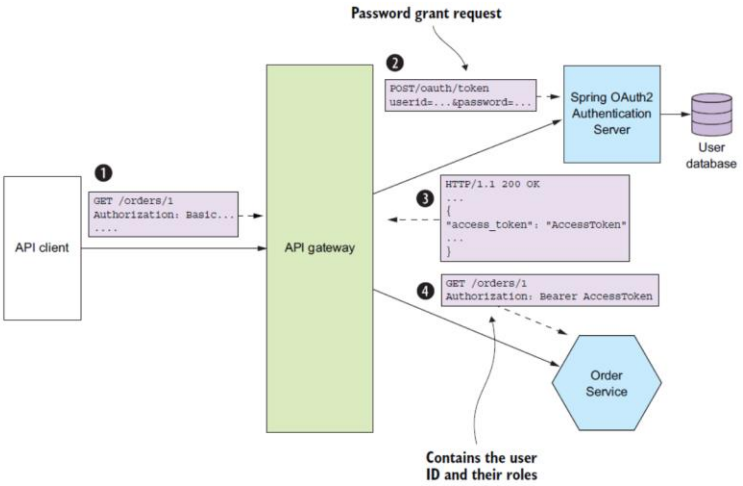
- Opaque Tokens, the recipient of the such token needs to make synchronous RPC call get the user information.
- Transparent Tokens, the most common type of token used for authorization. JWT token is one of widely used token that has a payload containing user information. The token itself can be used to get the authenticated user role. However, there is drawback of such token, JWT is a self-containing token Tokens that falls into malicious hand, does not have any mean to invalid the token unless the token is expire. This drawback can be overcome by issuing a short-live token, but then the system needs an efficient mechanism to re-issue token for each expiry token.

Using OAuth 2.0 for handling

Some key concepts of OAuth 2.0

- Authorization Server: Server that provides API for authenticating user, obtain access token and refresh token.
- Access Token: The token that provides access to the resource server.
- Refresh Token: long-lived yet revocable token, this clients usages to get new access token.
- Resource Server: The servers that use access token to authenticate its requester.
- Client: The client that access resource server.

How OAuth2.0 works



- API gateway is responsible for authenticating the client.
- API Gateway use access token to pass on the user (principal) information among services.
- Services leverage access token to identify the user (principal) identity & role.

Externalizing Configuration: there are two different ways to externalized configuration.

- Push Model – where the deployment infrastructure passes the configuration properties to the service instance – for example system / environment variable. Though its popular way to externalized configuration, but it difficult to reconfigure a service without restarting.
- Pull Model – where the service it self will reads the configuration property from the configuration server. The following are the ways to implement pull-based configuration externalization.
    - Version control such as GIT
    - SQL or No-SQL database
    - Using specialized config servers like – Spring cloud server , HarishCorp vault, AWS parameter store.

  Benefit of using a specialized config server
    - Centralized storage of all configuration in single location.
    - Transparent Decryption of the Sensitive Data – sensitive information can be encrypted and stored safely.
    - Dynamic Reconfiguration – service can be made to reconfigure easily without restarting.

  Drawbacks of using a specialized config server
    - Unless the specialized config server is provided by the deployment infrastructure , one need to manage one additional deployment.

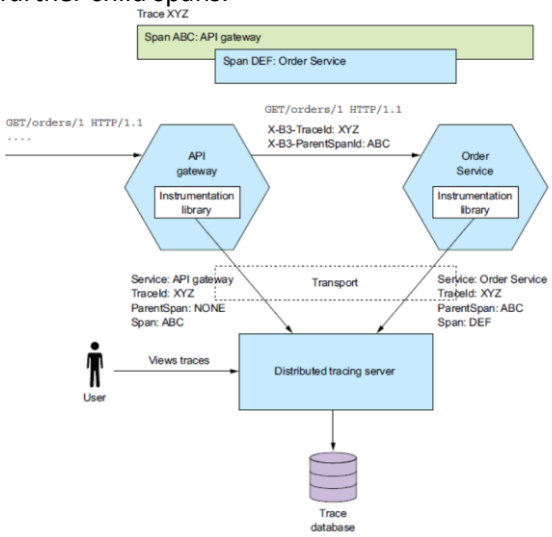
Observability

Common design observability design patterns

- **Health Check APIs.** API that are exposed that returns service health checks results. These API can also be configured with deployment infrastructure service (Kubernetes / docker) to determined if the service is up and running before routing workload. *One of the most common health monitoring tool is spring actuator.*
- **Log Aggregation**: Service that helps writing / aggregating logs into a centralized repository. Its not always a best practice to write logs to a permanent location, usually container & serverless applications don't have access to shared permanent location to write their logs,

thus its best to write the logs to sdout and let the deployment infrastructure to decide where to store/route the logs. *One of the most popular logging infrastructures is ELK (Elastic Logstasth, Kibana) stack.*

- **Distributed Tracing:** Assigned each external request unique individual IDs that can be leverage to track the (external) request throughout the system. For distributed system, tracing a unique external request can be cumbersome, thus maintaining a single unique id through out the transaction helps, at each stage (service) entry and exit time will be log, that will give further trace of latency request is facing at each individual service level. In distributed service terminology – trace is the external request, which will have one or more spans. Spans can help further child spans.



- **Exception Tracing:** Report / trace / de-duplicate exceptions that helps alerting developers and track the resolution for each of the exceptions. Tool likes Honeybadger (www
- . honeybadger.io) or Sentry.io (https://sentry.io/welcome/) can be leveraged to implement exception tracking.
- **Application Metrics**: various maintenance metrics that helps in identifying health of the system. Its important to capture, infrastructure as well as application-level metrics which then needs to be displayed through a unified metrics service. There are two different ways to share metrics data to metrics service – using push mechanism where service send the data to metric service endpoint or through pull mechanism where metrics service agent are deployed to the service which pull the metrics data and send it metrics service. Prometheus, is a popular pull mechanism where AWS Cloud watch is a popular push base metric service.
- **Audit Logging**: helps in customer support, compliance auditing requirements and also can be used to detect suspicious activity. This can be implemented using – adding audit logging code to the business service, or using spring AOP or using event sourcing.

Implementing cross cutting concerns can be cumbersome, A much faster approach is to build microservices using microservice chassis framework. A chassis framework will offload the flowing concerns from the code

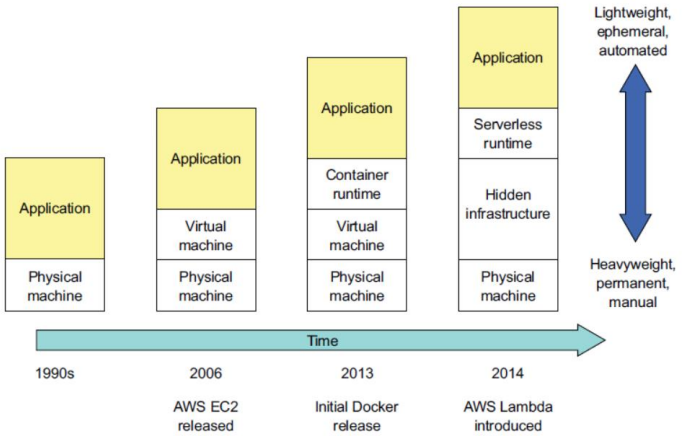
- Externalizing configuration.
- Health Check
- Application Metrics
- Service Discovery
- Circuit Breaker
- Distributed Tracing

One of the drawbacks of using microservice chassis is, many of the concerns that can be handled through infrastructure needs to be handle functionally within the microservice chassis framework.

Chapter 12: Deploying Microservice

When deploying a newer version of a code into production environment, the following things user needs to access functionalities implemented by those services.

- Service Management
- Runtime Service Management
- Monitoring
- Observability



Different deployment option

- **Deploy Services to a specific language – JAR/ WAR**: Multiple Service can be deployed to a single process for example multiple war files deployed to a single tomcat server, or each services deployed to separate process for example, each .war deployed to an separate tomcat server.

Advantage	Disadvantage
Fast Deployment	Lack of encapsulation for the technology stack.
Effective utilization of resources	No ability to contains resources consumed by a service. Lack of isolation for the services.
	Each of the services, capacity plan need to be guess correctly for effectively utilizing the available resources.

- **Deploying Service as virtual machine**. Services can be package as VMs, which can effectively used for deploying services in isolation.

Advantage	Disadvantage
Effectively encapsulate technology stack.	Less effective resource utilization.
Isolate services	Relatively slower deployment
Leverage mature cloud infrastructure	System administration overhead.

- **Deploying Service as container**. Deploying service in a container makes the services more manageable, as each of the container has its own ip and does not cause port conflict. Also, container orchestration tools like Kubernetes make the management of containers more effective by providing features like

- Resource management manging CPU, Storage, Memory etc.
- Scheduling selects machine (nodes) where the container needs to be running – can be configured to run in affinity or non-affinity mode.
- Service management features, K8 implements named and version services which can be directly mapped to each microservices, making sure that desired number of healthy instances are always running.

Advantage	Disadvantage
Effectively encapsulate technology stack.	Still needs to manage System administration overhead of the host system.
High degree of Isolation can be achieved.	
Resource per containers can be constrained.	

- **Deploying Service as serverless deployment.** *"Magic happens at the intersection of functions, events, and data." - Re:Invent 2014, Werner Vogels, the CTO of Amazon. Instead of deploying the container containing business functions, in case of serverless deployment the business functions is packed in jar/zip which then can be triggered by*
  - HTTP Request
  - Events
  - Scheduled invocation
  - Direct API call

Advantage	Disadvantage
Can natively integrated with Cloud Infrastructure services	Latency in trailing logs
Eliminates needs for managing / maintaining system administration tasks.	Limited event/request base model.
Highly Elastic – Capacity does not needs to be guess during the initial, based on the incoming load the services capacity can be automatically increases / decreases.	
Pricing based on actual usages of the resources.	

AWS Lambda function does not provides, spring feature like dependency injection however it's possible to use spring feature like dependency injection by creating a spring context then use the context to use spring features like dependency injections.

```
public abstract class AbstractAutowiringHttpRequestHandler
extends AbstractHttpHandler

{

private static ConfigurableApplicationContext ctx;

private ReentrantReadWriteLock ctxLock = new
ReentrantReadWriteLock();
```

```
private boolean autowired = false;

protected synchronized ApplicationContext getAppCtx()
{
ctxLock.writeLock().lock();
try {
if (ctx == null) {
ctx = SpringApplication.run(getApplicationContextClass());
}
return ctx;
} finally {
ctxLock.writeLock().unlock();
}
}
@Override
protected void
beforeHandling(APIGatewayProxyRequestEvent request, Context
context) {
super.beforeHandling(request, context);
if (!autowired) {
getAppCtx().getAutowireCapableBeanFactory().autowireBean(this);
autowired = true;
}
}
protected abstract Class<?> getApplicationContextClass();
}
```

This class overrides the beforeHandling() method defined by AbstractHttpHandler. Its beforeHandling() method injects dependencies using autowiring before handling the first request.

Subclasses to implement `getApplicationContextClass();`

```
public class FindRestaurantRequestHandler extends
AbstractAutowiringHttpRequestHandler {

  @Autowired
  private RestaurantService restaurantService;

  @Override
  protected Class<?> getApplicationContextClass() {
    return CreateRestaurantRequestHandler.class;
  }

.
.
.
.
```

AWS needs the java code to package as jar or zip file. Business services can be deployed as serverless function using SAM, serverless application module which then can be configured to AWS Gateway to make the services accessible through AWS Gateway endpoint. Business service code can be update by rebuilding the ZIP file.

Appendix

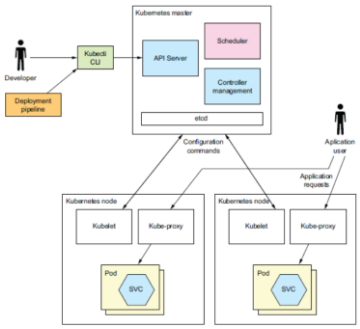
What are different types of scaling, explain scale cube?

Martin Abbott and Michael Fisher's in their book "*Art of Scalability*" defines scale cube. In scale cube there are three ways of application scaling defined.

- **X-Axis scaling: Load balance across multiple instances**. This is most common form of application scaling, where multiple instances are created behind a load balancer and the load balancer distributes the load across these instances.
- **Z-Axis scaling: Route request based on the attribute of the request.** Unlike X-Axis scaling where requests are forwarded to one of the available load-balanced instances, in case of Z-Axis scaling the specific request (base on request attribute) are routed to specific instances. For example, based on region the request is routed to specific instance which process request from the region.
- **Y-Axis scaling Functionally decomposing application into services**. X-Axis and Y-Axis solved the problem of availability and complexity but does not address the deployment complexity. In case of Y-Axis scaling each of the business functionality are divided into discrete services which can be scaled independently.

Describe Kubernetes Architecture?

Kubernetes is one of the famous Docker container orchestration tools, it has a large open-source community of users. Each of the Kubernetes deployment has one master node and one or more slave nodes. Within each node one or more pods runs, within each pods one or more container are deployed.



Within master node the following services are hosted

- ***API Services***: REST services to deploy and manage services user by `kubectl` command line interface.
- ***scheduler***: Scheduler service selects the node for running pods.
- ***etcd*** : etcd is a no-sql datastore to store cluster data.
- ***controller Manager:*** Controller manager ensures the state of the cluster is the intend state, it ensures desired number of healthy instances are running.

Within the (slave/worker) nodes the following services are hosted

- ***Kublets*** – it creates and manage pots within the node.

- ***Kube Proxy*** – manages networking including loadblancing within the pods.
- ***Pods*** – host one or more container containing application service.

Key important construct of the Kubernetes

- **Pods:** This are the basic unit of *kuberneties* deployment, where each pod usually host one container, in some scenarios its host more than one sidecare containers which implements supporting functionalities.
- **Deployment:** Deployment is a controller that ensures that desired number of instances are always available. Its supports advance feature like versioning, rolling update, rolling rollback.
- **Service:** Provides client with the service discovery feature. Each service are assigned DNS name, DNS name can be resolved to IP to which load balanced TCP/UDP traffic to a pod.
- **Config Map:** Name value pair collection to externalised configuration, based on the DNS name, the pod can refer its environment variable from the ConfigMap.

Service deployed to Kubernetes cluster can be Loadbalancer by an external Loadbalancer, however it would not provide greater level of configuration. In order to achieve higher level of control on the following, service mesh can be used.

- **Traffic Management**
- **Security**
- **Telemetry**
- **Policy Enforcement**

What is spring cloud contract framework, what its use for?

How reverse proxy works?

What is Transaction Template in springframe work?

Template class that simplifies programmatic transaction demarcation and transaction exception handling.

Typical usage: Allows for writing low-level data access objects that use resources such as JDBC DataSources but are not transaction-aware themselves. Instead, they can implicitly participate in transactions handled by higher-level application services utilizing this class, making calls to the low-level services via an inner-class callback object.

Can be used within a service implementation via direct instantiation with a transaction manager reference or get prepared in an application context and passed to services as bean reference. Note: The transaction manager should always be configured as bean in the application context: in the first case given to the service directly, in the second case given to the prepared template.

Supports setting the propagation behaviour and the isolation level by name, for convenient configuration in context definitions.

In the below example there are two separate transactions are been handled. First Transactions is used for creating an order the second transactions is used for validating an order.

```
@Test
public void shouldSaveAndLoadOrder() {

  Long orderId = transactionTemplate.execute((ts) -> {
    Order order =
            new Order(CONSUMER_ID, AJANTA_ID, CHICKEN_VINDALOO_LINE_ITEMS);
    orderRepository.save(order);
    return order.getId();
  });

  transactionTemplate.execute((ts) -> {
    Order order = orderRepository.findById(orderId).get();

    assertEquals(OrderState.APPROVAL_PENDING, order.getState());
    assertEquals(AJANTA_ID, order.getRestaurantId());
    assertEquals(CONSUMER_ID, order.getConsumerId().longValue());
    assertEquals(CHICKEN_VINDALOO_LINE_ITEMS, order.getLineItems());
    return null;
  });

}
```

How to implement springboot code in AWS lambda?

Steps

1. Create a new Spring Boot application using your preferred IDE or command-line tool.
2. Add the AWS Lambda dependency to your project using Maven or Gradle:

```
<dependency>
          <groupId>com.amazonaws</groupId>
          <artifactId>aws-lambda-java-core</artifactId>
          <version>1.2.1</version>
</dependency>
```

3. Create a new class that implements the RequestHandler interface from the aws-lambda-java-core library. This class will handle the incoming requests and return the response. For example:

```
@Component
public class MyLambdaFunction implements
RequestHandler<Request, Response> {
          @Override
          public Response handleRequest(Request request,
      Context context) {
                  // Handle the request using Spring
                  ApplicationContext applicationContext =
      SpringApplication.run(MySpringBootApplication.class);
                  MyService myService =
      applicationContext.getBean(MyService.class);
                  String result =
      myService.doSomething(request.getInput());

                  // Create and return the response
                  Response response = new Response();
                  response.setOutput(result);
                  return response;
          }
}
```

4. Configure your Spring application to handle the incoming requests by adding the necessary annotations and components. For example, you could use the @RestController annotation to define a REST endpoint and the @Service annotation to define a service bean that performs some business logic.
5. Build and package your Spring Boot application into a JAR file.
6. Upload the JAR file to AWS Lambda and configure the function to use the MyLambdaFunction class as the handler.

Once the above steps are completed, AWS Lambda function will be able to handle incoming requests using Spring.

What is Collections.singletonList() method used for?

How to design API with API-First Design.

URL : https://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10

With API first approach – designing the application keep target developer interest ahead of developer interests.

API-First Design is a series of best practices build across companies and industries that prioritized developer experience.

Committed to launch RESTfull webservices with RESTFull API marker language (RAML)

How API economy driving much of the total economy - API is the simple technical way to achieving transformations into business context by enabling

1. Go Mobile

2. Discover new review streams and models
3. Innovate

Key Axioms with API first approach:

1. Faster is better than slower: Agile is more than a management style, faster & smaller is better than Bigger & slower.
2. Developers are the new decision maker – ensure delivering better DX (developer experience).
3. KISS principle – keep it short & simple
4. Be Agile – shorter delivery cycle (t-minus 6months OR fail).
5. Design for what should be done instead of what can be done – during implementation time, evaluate if existing platform can support or if any compromised needs to made.

API first design step-by-step approach.

1. **Plan** – plan the final product.
2. **Design** – plan how the product would be and test its feasibility
3. **Lock the API Specification** – run multiple iteration of the API specification, fixed it and lock it down, auto generate API specification.
4. **Implement** – develop as per the locked API specification, test and validate.
5. **Operate & engaged** – Deploy the API > collect operation metrics > engaged with client developers > collated feedback > repeat the cycle based on the collective feedback.