Software Development Processes

- **Edge computing**: Decentralised, distributed computing infrastructure. Process data close to where it is being generated → low latency but limited computation power
- Cloud-computing: Running applications on resources managed by cloud providers without having to manage them ourselves
- Cloud-enabled: legacy systems designed for local datacentres, but modified to run on the cloud
- Cloud-native: Build and deploy in cloud environments
  - Microservices-based → easy to scale and deploy
  - Containers and orchestrators
  - DevOps/CI-CD

Deployment

- To consider: Portability, Interoperability, Availability, Performance, Maintainability, Security
- **Bare Metal**: Install and run the app directly on the physical hardare of a server without virtualisation or containerisation
  - ✓ Complete control
  - ✓ Physical Isolation → improved security
  - × Hard to cater to different platforms
  - × Potentially wasted hardware resources
- **Virtualisation**: Use hypervisor to emulate underlying hardware to create multple VMS, enabling multiple OS and applications to run independently on the same machine
  - ✓ Flexible, scalable
  - × Still a full OS running inside each VM
  - × Side-channel attacks
  - × Noisy neighbour problem: A resource-hogging app can impact the performance of other apps running on the same machine
- **Containerisation**: Instead of running multiple OS on a single machine, run a single OS using a container engine which runs multiple containers, each running an application. Used with an orchestrator which manages and automate the deployment, scaling and operation of containers
  - ✓ Lightweight: Containers only includes OS processes and the necessary dependencies → quick to launch
  - ✓ Write once, run anywhere
  - × Noisy neighbour problem
- **Serverless**: Run applications without managing a server by ourselves. Server management (e.g. resource allocation, adding new servers, OS updates) is done by cloud providers.

SDLC Models

- **Waterfall**:
  - Highly structured, following the 5 stages in sequential order: Specify requirements → Design → Implement → Test → Deploy
  - \+ Good for projects with well-defined, stable requirements
  - − Inflexible and resistant to new/changing requirements
- **Iterative**:
  - Develop through repeated cycles in small portions at a time
  - Depth-first: Focus on fleshing out specific components
  - Breadth-first: Evolve all major components in parallel
- **Agile (Scrum)**:
  - 4 main principles:
    1. Individuals and Interactions over processes and tools
    2. Working software over comprehensive documentation
    3. Customer collaboration over contract negotiation
    4. Responding to change over following a plan
  - Iterative development called sprints (2 – 4 weeks long) which allow quick responses to changes
  - At the end of each sprint, there will be a shippable product
  - Between sprints, review what has been done during the sprint
  - Daily scrum meeting
  - Product backlog:
    - Full list of things the software should do (i.e. requirements)
    - Requirements are ranked by priority
  - Sprint backlog:
    - Subset of requirements from the product backlog to be developed during a sprint

Software Delivery

- Deployment pipeline: automated build, deploy, test and release
- **Continuous Integration**
  - Automatically and frequently integrate code changes to single shared repository + automated build to detect problems early (prepare code for release)
- **Continuous Delivery**
  - CI + ensure software is always release-ready with manual checks
- **Continuous Deployment:**
  - CI + automatically release the product once tests pass
- **DevOps**: Blend software development and operations staff and tools
  - Using tools and automated processes to reduce the time it takes to test and release new changes while ensuring high quality
  - After deployment, perform application and network performance monitoring
  - Practices: CI/CD, Continuous Monitoring & Logging, Communication & Collaboration, Infrastructure as Code
  - 5 main principles (CALMS):

1. Collaboration between dev and ops team for shared responsibility
2. Automate repetitive tasks, reduce errors, increase efficiency
3. Lean strategy to minimize time to delivery
4. Measure performance by data collection and analysis
5. Sharing info and learnings across team

Requirements

- Usage-centric: User stories
- Product-centric requirements: Functionalities given by the product
- Development Phases:
  - **Elicitation**: Gather requirements (e.g. interviews, workshops)
  - **Analysis**: Analyze the practicalities of the requirements and **identify** and clarify gaps
  - **Specification**: Document the requirements for review
  - **Validation**: Confirm correct set of requirements with end-users
- **Software Requirements Specification (SRS)**: A technical document that lists all the requirements a software must satisfy, its functionalities, design and limitations
- Requirements should be under change control
- **Functional Requirements (FR)**: Specify software behaviour/function
- **Non-Functional Requirements (NFR)**: Quality + Constraints; describe how well the system works (e.g. reliability, efficiency, usability)
- **Businesss Requirements**: Why the organisation is implementing the software
- **User Requirements**: Goals/tasks the user must be able to perform with the software
- **Requirements traceability:** ability to follow the life of a requirement through its development to its deployment and use and periods of refinement
  - Use unique IDs to identify a requirement
  - Should be assigned a priority and brief high-level description
- **Requirement Validation**: Whether the requirements are correct
- **Requirement Verification**: Whether the requirements are implemented correctly (whether the specifications are met)
- Requirements can be documented in SRS, Product Backlog, User Stories etc.

Software Quality Attributes

- **External** QA: Observed when software is executing and impacts UX; the users' perception of software quality
  - E.g. Availability, Performance, Safety, Interoperability, Usability
- **Internal** QA: Not directly observed when software is executing but may impact external QA; the developer's perception of software quality
  - E.g. Efficiency, Scalability, Portability, Maintainability, Testibility
- Nature of the software dictates the importance of certain QAs (e.g. embedded software require performance and robustness more than other QAs)
- Security Requirements: should be decided by business owners
  - Authentication
  - Auditing and logging
  - Intrusion monitoring
- Safety vs Security:
  - Safety: Whether or not a system can bring harm (e.g. physical, financial)
  - Security: Concerned with privacy, authentication and data integrity
- Performance: primarily about responsiveness of system
  - E.g. response time, throughput, data capacity, behaviour in degraded modes, predictability in real-time systems
  - Can impact safety in real-time systems
  - Requirements often affect choice of architecture, design and deployment
- Scalability: ability to accommodate a growth in application usage
  - Hardware: Adding disk capacity
  - Software: Architecting to use parallel and/or distributed computing
  - Horizontal scaling: Add more machines; better for micro-services
    - \+ Increase resilience/fault tolerance
    - − Adds costs and complexity
  - Vertical scaling: Increase the capability of single machines
    - \+ Easier to maintain
    - − Single point of failure
- Availability: Measures the planned uptime of the system
  - Impacts the costs of deployment and complexity of software design
- Usability: User-friendliness, ease of learning and ease of use

Software Architecture

- Software architecture = the structures of the system comprising of software components, externally visible properties of those components, and the relationship between them
- Building blocks of an architecture:
  - **Component**: Element that models a specific feature
  - **Configuration**: Topology or structure
  - **Connector**: Element that models interactions between components
- Reference Architectures: A "template" high-level architecture for similar applications; lead to architectural patterns
- **Control Flow**: Represents computation order (C1 → C2 means that C2 follows C1)
- **Data Flow**: Based on data availability ; represents how data flows through a series of computations
- **Call and Return**: Control moves from one component to another and back
- **Message**:
  - Some data sent to a specific address
  - Each component awaits messages and reacts to them
- **Event**:
  - Some data emitted for anyone to listen to

Architectural styles

- Technical partitioning
  - Focus on separation of concerns
  - Align with teams' expertise (e.g. [Presentation, Services, Persistence] aligns with [Frontend, Backend, Database] expertise
- Domain partitioning
  - Logical components map to the problem (e.g. [Customer, Shipping, Payment]
- Monolithic deployment: Deploy all logical components as one unit
- Distributed deployment: Logical components run as individual processes and communicate over the network

Layered Architecture (N-tier)

- Monolithic deployment, techinical partitioning
- Higher layers make use of services provided by lower layers; lower layers are independent of higher layers
- Supports independent and evolution of different system parts
- Open-layer: Direct communication can happen between non-contiguous layers
- Closed-layer: A layer can only directly communicate with the layer below it

Pipe-and-Filter

- Data enters the system and flows through components one at a time until the data is assigned to some final destination (data sink)
- Each component (filter) transforms the data before passing it to the next component
- There should be some common representation of the data among each of the filters
- Filters are indeoendent of one another

Model View Controller (MVC)

- View: Displays data, interacts with users and pulls data from model
- Controller: Detects and responds to UI events and updates the model/view; coordinates between model and view
- Model: Stores and maintains data/business logic; updates view
- **Web MVCs**
  - View; renders the HTTP response returned by the Controller , possibly using templates (View updates the display instead of Controller)
  - Controller: handles HTTP requests, select the model and perpares the view
  - Model: business logic + persistance (database)

Single Page Applications (SPAs)

- Multi-page applications: When client sends a HTTP request which causes an update to the view, server returns an updated HTML file. A page reload is triggered to get the updated view
- In SPAs, server returns JS along with the initial HTML on first load. When client sends a HTTP request which causes an update to the view, server only returns the data (e.g. JSON text). Client runs JS to update the view according to the data

Microservices Architecture

- A single application comprised of smaller well-defined (cohesive), independent services which communicate with one another through well-defined mechanisms
- **Requirements of a microservice**:
  1. Each microservice should **be developed and deployed independently**
     - Each service is owned by different teams
     - Their implementation should be hidden from other services
  2. Each microservice offers a **well-defined business capability**
     - Its scope is closely aligned with business capabilities
     - Is cohesive: Features inside it must be highly related
  3. Microservices communicate through well defined mechanisms

Domain Driven Design (DDD) methodology:

- Focuses on modeling software based on the business domain
- Divide domain (problem space) into smaller sub-domains (problem space) and bound the latter in cohesive boundaries
- E.g. Domain: E-commerece. Subdomains: Product Purchase, Product Inventory, Financial Accounting etc
- **Bounded context**: solution space for sub-domains
- Interaction between contexts model interactions in sub-domains
- Identify collaborations between bounded contexts
  - **Shared kernel**: 2 independent contexts overlapping some subnet of each other's domains (heavy coupling)
  - **Upstream-Downstream**: Upstream context produces something for the downstream context to consume (like pub-sub)
  - **Supplier-Customer**: Supplier provides agreed-upon interfaces (API) to deliver the expected functionalities to Customer. Customer may influence the features of the interface to ensure it meets their needs
  - **Comformist**: Comformist context adapts to the model of another context, without attempting to influence any change
- Interactions between bounded contexts can be synchronous (e.g. RESTful APIs over HTTP) or asynchronous (e.g publish-subscrib
- **Aggregate**: a smaller context; cluster of **related objects** (instead of subdomains) to be treated as a single unit, for the purpose of data changes
  - Transactional boundary: Any changes to the aggregate either all succeed or non succeed
  - Consistency boundary: objects outside the aggregate can only read the aggregate's states and modified using the aggregate's root (e.g. a driver does not have to individually control each wheel of a car. A car is an aggregate of several other objects like the wheels, engine, brakes etc)
  - Aggregate Root: Parent entity of the cluster which acts as the aggregate's public interface
- Bounded contexts/aggregates can be used to identity microservice boundaries

Event Storming

- **Domain events** = events that happened in the past in the domain; typically involving state changes (e.g. "Order placed", "Payment processed")
- **Command**: Something (user or external system) that triggers domain events (e.g. "Place order, "Process payment")
- **Aggregate**: Cluster of related domain objects that handles incoming commands, then validates and updates state, and outputs the correct events (e.g. "Place order" command → "Order" aggregate → "Order placed" event)
- **Policy**: business rule or constraint that govern the flow of events; listens for events then triggers a command; WHEN *event* THEN *command* (e.g. "When *payment card is submitted*, then *confirm order*")
- **Boundary context**: Cluster of related aggregates; boundary contexts are linked by policies
- Boundary contexts can then be used to identify microservice boundaries

Microservices Design Patterns for Data Isolation

Database-server-per-service pattern

- Each microservice has its own database

\+ Loose coupling and allows independent scaling at database level

\+ Easier to replace underlying database technology to something appropriate with each service

− Explosion of number of database clusters; expensive to maintain

Private-tables-per-service pattern

- Each microservice owns a set of tables that is only accessible by that service

Schema-per-service pattern

- Each microservice has a database schema that is private to that service

Data Delegate pattern

- If multiple services need to access shared data from a table, design a delegate service that sits in between the services and the database
- The services can depend on the delegate service to access/modify data in the table. Only the delegate service has access to the table

Data Lake pattern

- Data lakes: read-only, query-able data sinks containing a copy of data (typically data-change events) from all concerned microservices
- Related microservices read from the data lake instead of querying from other microservice's databases directly

Saga pattern

- Handles long-running, distributed transactions involving multiple microservices, to ensure data consistency across the involved microservices
- Used in event-driven architecture
- At every step of the transaction, we define a **compensating action** which is to be executed if we happen to rollback the transaction later
- Does not follow ACID properties: does not promise rollback to initial state
- Sequence does matter; better to move steps that are harder to compensate to end of transaction

Event sourcing

- Events are sources of truth and stored (appended) in an event store (e.g. Kafka); state is derived from the sequence events
- Events act as both data and means of async communication between services
- An event is represented by
  - Unique event id
  - Event type (e.g. "priceIncreased")
  - Relevant data for that event type (e.g. "amount", "currency")
- **Projections** are functions that calculate new state, given the current state and an event
  - Chain projections to calculate the state from a sequence of events
- Recalculating state from the beginning can be very expensive; to mitigate, use **rolling snapshots** to calculate and save intermediate states, and calculate the state from the last snapshot
- Event stores need to support
  1. Store new events and assign correct sequence
  2. Notify event subscribers who are building projections about new events they care about
  3. Get N number of events after event X for a specific event type, for reconciliation flows

Command Query Responsibility Segregation (CQRS) pattern

- Events sourcing is usually paired with CQRS pattern
- Separates read (query) and write (command) operations into distinct models (command and query models)
  - E.g. New events are appended to the event store like Kafka. A component is that is subscribed to the event store listens for new events and calculates new state using projections and store the new state ("view") in some db. To read data, query from the db

\+ Read and write can be optimised and scaled independently

− Read data may be stale (if read is performed immediately after write), but there is eventual consistency

Microservices Design Patterns for Indep. Deployment

- Each microservice has its own specific deployment, resource, scaling, and monitoring requirements. Each service must be provided with the appropriate CPU, memory and I/O resources
- **Infrastructure as Code (IaC)**
  - Application's underlying infrastructure can be more easily versioned using code
  - Writing and executing code to define, deploy, update and destroy
- **Immutable infrastructure**
  - For consistency, predictability and easier rollbacks
  - Changes must be made by re-creating the component

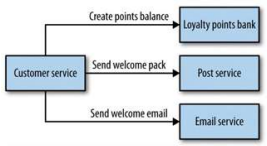
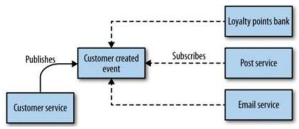
Multiple-instances-per-host pattern

- Multiple microservices running on a single host, on different ports
- Typically for monolithic architecture

Service-instance-per-host pattern

- Run each service instance in isolation on its own host
- Per-VM or per-container

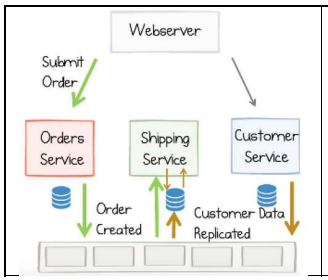
Service Collaboration Patterns

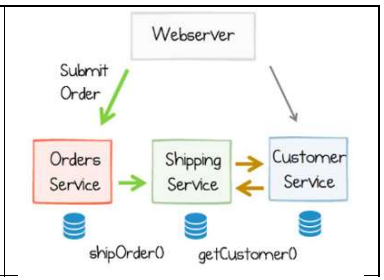
Orchestration	Choreography
	
Centralised control; rely on a single service to manage and coordinate interactions between services	Decentralised control; services communicate and react to events independently
Used in complex business processes that need a specific sequence of operations	Used in real-time data processing
Single point of failure	Loose coupling provides fault tolerance
Less scalable as orchestrator becomes the bottleneck	Highly scalable
Easier error handling	Complex, distributed error handling
Less flexible	Highly flexible and adaptable

Service Communication Patterns

Pattern	Definition	Use Cases
Request-Sync Response	Client sends a request to a service and waits for response	API calls, synchronous processes
Notification	Client sends a request to a service but does not expect a reply	Logging
Request-Async Response	Client sends a request to a service and receives a response later (via callback or polling)	Long-running processes e.g. file uploads, payment processing
Event-Driven	Services react to events as they occur	Real-time systems, pub-sub

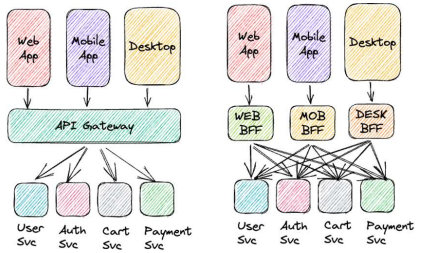
Event-Driven	Request-Response
Asynchronous	Generally synchronous
Loose coupling	Tight coupling as requester needs to know details of the responder



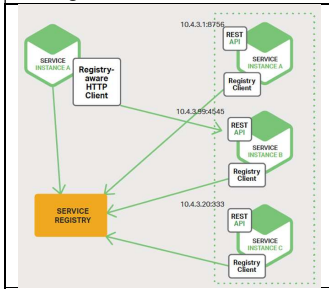
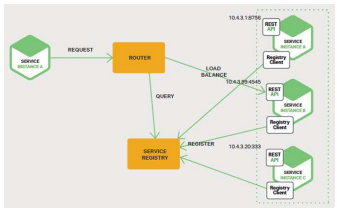


Service Discovery Patterns

- **API Gateway**: A single entry point into the system for the client; it encapsulates the internal system architecture and exposes an API for the client
- **Backends For Frontend**: Variant of API gateways; essentially multiple API gateways, one for each client, to have a tailored API that targets the needs of each client (e.g. mobile, web)



- **Service registry**: A database of services, their instances and their locations. Service instances are registered with the service registry on startup and deregistered on shutdown

Client-side Discovery	Server-side Discovery
Client queries the service registry, then uses a load-balancing algorithm to select an available service instance	Client makes a request to a service via a load balancer, which queries the service registry and routes each request to a service instance
More complex client	Simplfies the client
Changes in service instances may affect multiple clients	Single point of failure
More control to clients for advanced routing	Centralised control and easier for dynamic scaling
	

Event Driven Architechture

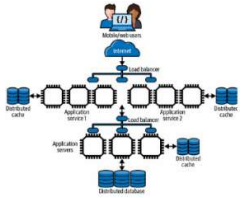
- Events:
  - Initiating event: originates from an end user; typically kicks off a business proceccs
  - Derived event: internal events generated in response to the initiating event
  - Unkeyed: Describe a singular statement of face
  - Entity: Unique thing and is keyed on the unique ID of that thing
  - Keyed: Contains a key but does not represent entity; used for partitioning the event stream to guarantee data locality within a single partition of an event stream
- **Asynchronous communication**: When an event is broadcasted, it does not wait for a response nor care whether other componenets are available or not
  - \+ Bettter responsiveness; no need to wait
  - \+ Better availability
  - − More complex/fragmented error-handling (relies on saga pattern) than in synchronous
- Components in EDA:
  - **Event producers** publish data onto event streams
  - **Event consumers** listens for and consumes the event data
  - **Event brokers** help manage the event log
- Event driven microservices (EDA + Microservice):
  - Requirements:
    1. Mostly async communication (EDA) via events
    2. Cohesive services (MS)
    3. Data ownsership (MS)
  - Microservices communicate by producing and consuming events

Event Brokers

- Receive, stores and provides events for consumption
  - Queues: Event is removed once it is read by a subscriber (hence can only be read by one subscriber)
  - Partitions: Event stays in the partition after being read
- Event log is immutable, append-only, preserving the state of event ordering
- **Write**: Append an event to an event log in a parition
- **Read**: Seeks to the index-ID of the last message it read (i.e. offset) in the partition, then scans sequentially from there while periodically recording its new index-ID in the partition
- **Kafka**:
  - **Topic**: A named channel where publishers send events (e.g. "user_signups" topic contains events relating to user signups)
  - **Partition**: Each topic is partitioned (based on event key) into smaller groups (e.g. events related to a specific user go to the same partition based on user ID)
  - **Consumer groups**:
    - Consumers are grouped into consumer groups
    - Each consumer within a consumer group reads from exclusive partitions (i.e. no partition can be read by > 1 consumer from the same consumer group). Hence, # consumers in consumer group ≤ # partitions
    - Consumers in a consumer group read in parallel

Scalability

- To increase capacity in some application-specific dimension (e.g. # requests, # data)
- Horizontal Scaling (Replication)
  - **Load balancer**: To choose a server Replica to process the request
  - **Session store**: To remember Sequence of user interactions and session state, because load balancer must be free to send requests from same client to different service instances; this helps to ensure that each server instance remains stateless

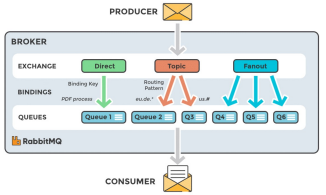


  - **Caching**: Caching to prevent frequent db lookups
  - **Scale the DB**
    - **Scale out DB Read Replicas**: Distributed read-only DB to handle many read requests at low latency. Use main database node for writing
    - **Database Partitioning**: Horizontal (rows) vs Vertical (columns)
  - Scale processing with multiple tiers
    - Services call one or more dependent services which in turn are replicated and load-balanced
  - Replicate entire application for different platforms
    - Each replica then utilize the core services that provide db access
- Vertical Scaling
  - Optimise existing resources (e.g. faster algorithms)
- Swim Lanes (Pod architecture)
  - Place a group of services within a boundary such that any failure within that boundary is contained within the boundary and the failure does not propagate or affect services outside of the said boundary

Messaging Patterns

Message	Event
Command or query to perform some action or to get some data	Carries information which is a fact; something has happened in the past
Point-to-point (p2p)	Can be broadcasted (pub-sub)
Uses queue which has a single receiver	Uses topic, enabling one-to-many communication
Queue: FIFO	Topic: Allow non-sequential processing via partitions
Receiver owns the message payload	Publisher owns the message payload

- **Advanced Message Queuing Protocol (AMQP)**
  - Brokers that use AMQP maintain **exchanges** and **queues**
  - Messages are published to exchanges with handle routing of the messages (helps decouple the routing from the client)
  - Exchanges distribute message copies to queues
  - Brokers either deliver the messages to subscribers, or the subscribers pull messages from queues on demand
- Exchange types:
  - **Direct**: Message is routed to queues whose binding key exactly matches routing key of the message
  - **Fanout**: Route messages to all queues bound to it
  - **Topic**: Route messages to queues whose binding key is a routing pattern that matches the routing key of the message

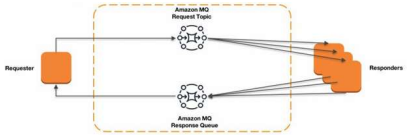


Message Construction

- **Header**: Contains metadata e.g. origin, destination, size
- **Properties**: Optional, often used for message selection and filtering
- **Body**: Contains the actual payload

Message Channels/Queue

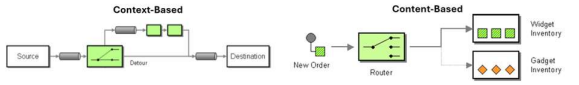
- A message channel is **UNI-DIRECTIONAL**. Two-way messages need 2 channels: Request and Reply channel
- Message channels are P2P (i.e. one-to-one)
- Correlation ID: Reply message contains correlation ID which is equal to the message ID of the message it is replying to
- Return address: Request message contains the return address which specifies which channel the replier should send the reply message to
- **Invalid Message Channel**: Erroneous messages are sent to this channel (e.g. Receiver receives request to delete a db record with some ID, but the ID does not exist)
- **Dead Letter Channel**: Contains messages that could not be delivered (e.g. due to network issues or invalid destination address)
- **Datatype Channel**: For specific type of data
- **Pub-Sub Channel**: One-to-many; any message published to a topic is received by all subscribers of the topic



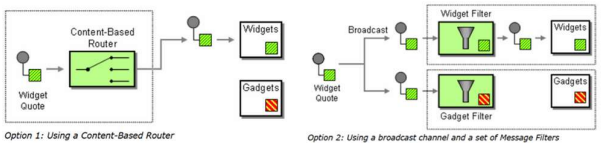
  - Note: Pub-sub channel is still one-to-one. When the subscribers send their responses back to the publisher, these responses need to be aggregated and sent to a P2P channel back to publisher

Message Router

- Simple Router: Route messages from one inbound channel to one or more outbound channels
- Composed Router: Combine multiple simple routers to create more complex routing logic
- **Context-Based** Router: Decide destination based on contexts rather than payload, e.g. environment, headers. Typically used to perform load balancing, test or failover
- **Content-Based** Router: Decide destination based on payload



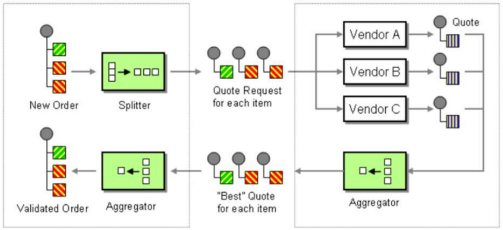
- **Message Filter**: Singe output channel; filters messages whose content matches the criteria specified by the message filter



Content-Based Router	Pub-Sub with Message Filter
Exactly one consumer	> 1 consumer may consume
Central control and maintenance, predictive routing	Distributed control and maintenance, reactive filtering
Router needs to know about participants	Router does not need to know about participants
Often used for business transactions	Often used for event notifications
More efficient with queue-based channels	More efficient with pub-sub channels

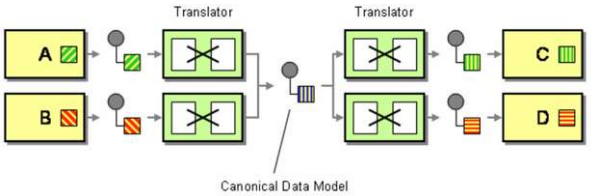
Message Splitter & Aggregator

- **Splitter**: Splits a single message into multiple messages (e.g. split an "order message" into multiple "order item message")
- **Aggregator**: Combine multiple messages into a single message (using correlation ID)
- **Scatter-Gather**: Broadcast single message, then aggregate the replies to a single message



Message Transformation

- Message translator between client and service converts messages from one format into another
- Helps decouple services; services do not care who receives the message and thus do not need to know about each other's data formats
- **Canonical Data Model (CDM)**: Two message translators between each pair of services (A to B and B to A). If there are n services, there will be n(n – 1) translators, which is too many. Instead, every service's translator translates its data into a canonical format. At the receiving service, the translator translates from canonical format to the service's own data format.

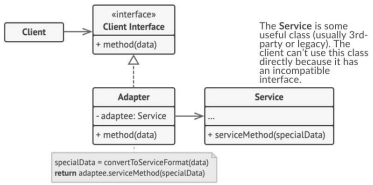


Message Endpoints

- An endpoint is the interface between an application/service and the messaging system
- The rest of the application knows little about the messaging details
- One endpoint, one message channel
- An endpoint can either send or receive messages, but not both
- **Polling Consumer**: Consumer controls when it consumes messages from the channel; it is proactive
- **Event-Driven Consumer**: Consumer cannot control when it consumes as it consumes the message as soon as it arrives; it is reactive

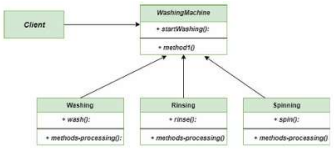
GOF Design Patterns

Adapter Pattern



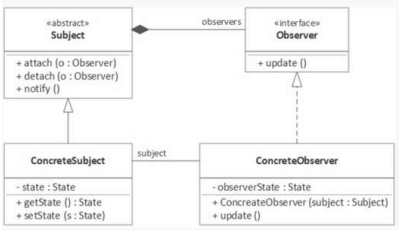
- Reconcile differences between 2 incompatible interfaces
- The Adapter is a class that implements the client interface while wrapping (contains) the service object
- The Adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand

Façade Pattern



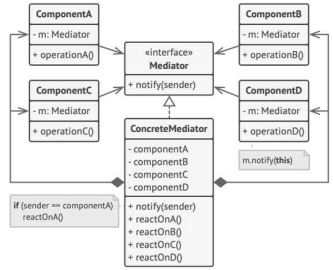
- The Façade provides a simple unified interface to a set of interfaces in a subsystem
- Clients interact with the façade instead of the individual interfaces
- The API gateway in Micro-service Architecture acts as the façade to all the individual microservices

Observer Pattern



- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- "Pub-sub" pattern in event-driven architecture
- **Subject Interface**: For observable objects
  - The interface should specify a method for Observers to register themselves as Observers to the Subject (e.g. registerObserver) as well as a method to notify all Observers when state changes (e.g. notifyObservers)
  - Subjects maintain a list of Observers
- **Observer Interface**: For observer objects
  - The interface should specify a method to update the Observers in response to a change in the Subject's state (e.g. an update method that is called in notifyObservers by Subject)

Mediator Pattern



- Define a Mediator object that encapsulates how a set of objects interact.
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly and lets you vary their interaction independently
- **Mediator Interface**: Implemented by the concrete Mediator

Data Transfer Object (DTO) Pattern

- DTO: A group of values In an ad-hoc structure just for the purpose of passing data around
- Reduces roundtrips by batching up multiple params in a single call → reduces network overhead