CS3219
AY24/25 Sem 1
github.com/jasonqiu212

01. Introduction

Software Delivery

- **Deployment** - Make software available to use after dev.
  - Bare metal: Customized build for target platforms
  - Virtual machine: Use VM to run guest OS to run app.
  - Containers: Include only necessary OS processes and dependencies (Lighter than VM)
  - Serverless: Cloud-native servers that don't need developers to manage (Let provider manage resources)
- **DevOps** - Practices combining software dev. and ops.
  - Purpose: Reduce time between committing change to the change reaching production while ensuring quality
  - **Cont. Integration** - Auto build, unit test, deploy to staging, and acceptance test, to show problems early
  - **Continuous Delivery** - Same as above, except with manual deployment to production. Ensures that every good build is potentially ready for production release.
  - **Continuous Deployment** - Same as above, but with auto deployment to production

02. Requirements

- Capabilities needed by user or must be met by system

Requirement Types

- **Business Req.** - Why the org. is implementing the system, e.g., reduce cost by 25% (**Vision and scope doc.**)
- **User Req.** - Goals the user must be able to perform with the prooduct, e.g., check for flight on website
- **Functional Req. (FR)** - Specifies what a system does, e.g., website can export boarding pass
- **Non-Functional Req. (NFR)** - Not directly related to functionality of system, e.g., how well it works
- **Business Rules** - Policies that define or constrain requirements, e.g., staff gets 40% discount
- **Quality Attributes** - How well the system performs, e.g., Time bet. failure $\geq$ 100 hours. Type of NFR.
- **System Req.** - Hardware or software issues, e.g., invoice system must share data with purchase order system
- **External Interfaces** - Connections between systems and outside world, e.g., must import files in CSV format
- **Constraints** - Limitations on implementation choices, e.g., must be backward compatible. Type of NFR.

Requirements Development Phases

- **Elicitation** - Discover requirements (e.g., Interview)
- **Analysis** - Analyze, decompose, derive, understand
- **Specification** - Written or illustrated requirements
- **Validation** - Confirm correct set of requirements
- No linear path

Requirements Development Outcomes

- **Software Req. Specification (SRS)** - Complete desc. of behavior of software. Contains FRs, System Req., Quality Attributes, Ext. Interfaces, and Constraints.
- **Rights, Responsibilities, and Agreements** - All stakeholders confident of development within balanced schedule, cost, functionality and quality
- Requirements under **Change Control**

Quality Attributes

- Pros and cons = Good and bad of each attribute
- Quality attributes impact each other (Trade-offs)
- **Validation** - Do you have the right requirements?
- **Verification** - Do you have the requirements right?

External

- Impacts user's experience
- **Safety** - Whether system can do harm
- **Security** - Privacy, authentication, and integrity
- **Performance** - Responsiveness of system. Impacts safety for real-time systems.
- **Availability** - $\frac{\text{Up time}}{\text{Up time}+\text{Down time}}$
- **Usability** - User-friendliness and ease of use
- **Robustness** - How app performs when faced with invalid inputs, defects, and attacks
- **Reliability** - Probability of app executing without failure
- **Integrity** - Preventing information loss and preserving data correctness
- **Interoperability** - How readily system can exchange data and services with other software and hardware
- Others: Deployability, Compatibility, Installability

Internal

- Perceived by developers and maintainers
- **Scalability** - Ability to have more users, servers, etc.
  - Vertical Scaling: Add capability of machines
  - Horizontal Scaling: Add more machines
- **Verifiability** - How well software can be evaluated to demonstrate that it functions as expected
- Others: Maintainability, Testability, Modifiability, Portability, Reusability, Efficiency

03. Software Architecture

- Contains: Components, Connectors, Configuration
- **Reference Architecture** - Common architectural framework that leads to architectural patterns
- **Control flow** - Connector indicating computation order
- **Data flow** - Connector indicating data flow
- **Call and return** - Control flow moves from 1 component to another and back
- **Decomposition** - Breaking down a system
  - **Horizontal Slicing** - Designing by layers
  - **Vertical Slicing** - Designing by features

Architectural Styles

- Categories: How is code divided? (Technical/domain partitioning), How is system deployed? (Mono./distri.)
- **Monolithic** - Good for small apps; Faster dev., testing, deploy., maintainability, performance; Cheaper infra.
- **Distributed** - Good for complex apps; Scalable, decoupled, fault isolation, maintainability
- **Layered** - Software organized as layers of components that communicate via interfaces
- **Pipe and Filter** - Data flows through components (Data source → Filters → Data sink) via pipes
  - Pros: Modular, Flexible, Extensible, Scalable
  - Cons: Stateless, Data format, Comm. overhead
- **MVC** - Model (Business logic), View, Controller (Coordinates between view and model)
  - Cons: Coupled controller, Complexity, Maintainability
- **Web MVC** - 2 communicating entities: Server (Holds the model) and Client (Interacts with the server)
  - Controller: Handles user HTTP request, selects model, prepares view
  - Client-side Rendering (CSR): Rendered in browser with slower initial load but faster page changes
  - Server-side Rendering (SSR): Rendered in server with faster initial load but requires more server resources
- **Single Page App. (SPA)** - Implementation of CSR; retrieve data from server without refreshing single page

Representational State Transfer (REST)

- Set of rules for transferring, accessing, and manipulating textual data representations of hypermedia
- **Hypermedia** - Combo. of multimedia and hyperlinks
- Pros: Less coupled, Scalable, Interoperable
- Cons: Being stateless decreases network performance, URI degrades efficiency

Constraints

- Client-server architecture: For separation of concerns
- Stateless: Interaction between client and server should contain all information for scalability and reliability
- Cacheable: Server response should include if data is cacheable or not for network efficiency
- Layered: To reduce system complexity
- Uniform interface to interact with server (HTTP/S)
  - Resource-based: Anything can be a resource
  - Resources identified and manipulated through unique resource identifiers (e.g., HTTP DELETE /user/:id)
- Code-on-demand: Optional; Allow client functionality by downloading executable code

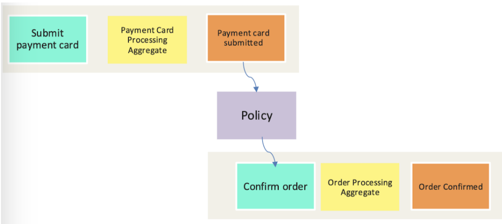
04. Microservices Architecture

- **Microservices App.** - App. as suite of small services
- Each microservice has well-defined business capabilities and cohesive features, is developed/deployed independently, and communicates with each other through well-defined mechanisms (Sync./Async.)
- How to identify boundaries of microservices? DDD and Event Storming

Domain Driven Design (DDD)

- Complex system is collection of multiple domain models (sub-domains)
- **Domain** - **Problem space** that business occupies
- **Sub-domain** - Component of main domain
- **Bounded Context** - Cohesive boundary in the **solution space** relevant to the sub-domain that helps to define the models, functionalities, and implementation needed
  - Shared kernel: 2 contexts developed independently but overlaps (Tightly coupled teams)
  - Upstream-downstream: 2 contexts in provider-consumer relationship through API
  - Conformist relationship: Consumer conforms entirely to provider (Most loosely coupled between teams)
  - Interactions between bounded contexts model interactions between sub-domains
- **Aggregate** - Cluster of related entities and objects that are **part of bounded context**, ensuring consistency and integrity through:
  - Transactional boundary: Any change to aggregate will either all succeed or none will succeed
  - Consistency boundary: Everything outside of aggregate can only read; state can only be modified through aggregate's public interface
  - Aggregate Root: Aggregate's public interface
- E.g. Bounded context: Order Management; Aggregate: Order, Customer

Event Storming



- **Command** - User or external action that causes events
- **Aggregate** - Unit for purpose of data changes after command and before event
- **Domain events** - Relevant events that occur in domain
- **Policy** - Relationship where event triggers command
- Bounded contexts determined by grouping commands, aggregates, and events; Policies link contexts

Data Patterns in Microservices

- Motivation: How do microservices manage data?
- **Database-server-per-service Pattern**
  - Data Indep.: Services should not modify same data
  - Pros: Loose coupling, Easy interoperability
  - Cons: Lots of DBs, Expensive
  - Private-tables-per-service: Service owns private tables
  - Schema-per-service: Service has private DB schema
- **Delegate Pattern** - Access DB through authoritative delegate service and avoid accessing DB directly. Pros: SoC, Extensible. Cons: Complexity, Performance
- **Data Lake Pattern** - Aggregate data from microservices into read-only, query-able data sinks. Pros: Democratized access, Decouples storage and processing. Cons: Performance, Security, Data governance
- **Sagas Pattern** - All steps have a compensating action that's stored on routing slip and passed along
  - If step fails, roll back using routing slip and revert to **reasonably** compensated state (e.g., notification)
  - Harder-to-compensate steps should be at the end
- **Event Sourcing** - Store stream of facts/events that got app. into current state, instead of storing current state
  - Event: UUID, Event type, Data relevant to event type
  - **Projection function** - Calculate new state using current state and new event
  - **Rolling snapshots** - Save projections to speed up perf.
- **Command Query Responsibility Segregation (CQRS)** - Split commands (write) from queries (read data)
  - E.g. Write into Kafka queue of events (Event Sourcing); Read from materialized view derived from events
  - Pros: Single write model can add data into many read models, Scalable

More Patterns in Microservices

- **Service Instance per Host** - Run each service instance in solation on its own host (e.g., VM, Container)
- **Immutable Infrastructure** - Component changes must be made by recreating component
- **Infrastructure as Code** - To easily version infra.
- **Orchestration** - Rely on central brain to drive processes. Cons: Single point of failure, Scalability and performance bottleneck
- **Choreography** - Inform each component of its job, and let it work out the details. Cons: Complexity
- Service communication: Sync/async? 1-way or 2-way?
  - **Event-Driven Communication** - See EDA
  - **Request-Response Communication** - Sync. request and waits for response. Cons: Latent, Coupled
- **API Gateway** - Entry point server that routes requests to services
  - Backends for Frontends: Gateway for each device type
- **Service Discovery** - Service registry to store IP and port of each microservice
  - **Client-side Disc.** - Client determines location from registry and uses load-balancing to select
  - **Server-side Disc.** - Client req. to router/load balancer
  - **Service Registry Pattern** - Database of services and locations, where instances register with registry

05. Scalability

- Scaling services in monolithic applications:
  - **Scale Up** - Upgrade server
  - **Scale Out** - Run multiple instances/replicas
    - **Load Balancer** - Chooses instance to execute req.
    - **Session Store** - Stores user's session across replicas
- Scaling databases:
  - **Caching** - For data with freq. read and rare writes
  - **Scaling Out with Read Replicas** - Write to primary and read from secondary to separate read and write
  - **Scaling Out by Partitioning Data** - Horizontal (By rows) vs. Vertical (By columns)
  - Scaling out databases creates **distributed databases**
- Scale multiple services to build multi-tiered apps.
- **Pod Architecture** - (Swim lanes) Place group of services/replicas inside boundary to contain failures
- Scale Cube: Run multiple instances/copies (X-axis), Split functionalities (Y-axis), Split data (Z-axis)

06. Event-Driven Architecture

- **Event** - Broadcasted to services that smt. **happened**
  - **Initiating Event** - From end user
  - **Derived Event** - Internal event due to initiating event
  - Structure: Key-value pair (**Unkeyed** - No key; **Entity** - Unique key; **Keyed** - Key not unique; For partitioning)
  - Publisher owns event payload and topic channel
- **Event-Driven Architecture (EDA)** - **Event-based** with **async.** communication
  - **Real-time data** - Published as it is generated
  - Components: Producers, Brokers, Consumers
  - Hybrid event-driven microservices, since micro. usually relies on sync. comm. via REST
  - Pros: Fast, Scalable broker, Less coupled
- **Broker** - Receives events, stores events in queue/partitions, and provides events for consumption (e.g., Kafka)
  - Properties: Immutable, Ordering, Indexing, Partitioning, Infinite retention, Replayability
  - **Partition** - Indexed queue that **persists** after pop
  - Consumer consumes by index of last message it read
  - **Topic** - Category of partitions; channel for **1-to-many** communication (Pub-Sub)
    - Multiple partitions → Non-sequential processing

07. Asynchronous Communication

- Communication types: Sync./async.? Single/multiple receivers? Persistent/transient?
- **Synchronous** - Caller sends message and **waits** for receiver to respond with ack. (e.g., Request-Reply with HTTP/S and REST)
- **Asynchronous** - Caller sends message and continues executing code without waiting (e.g., AMQP, Pub-Sub)
  - Pros: Responsive, Available, Decouples sender and receiver
  - Cons: More complex error-handling
- **Message** - Carries **point-to-point** (1-to-1) command or data query to be executed by another service
  - vs. Event: Both for async. communication, but with different intent
  - Receiver owns message payload and queue channel
- **Queue** - FIFO channel with **single receiver** (P2P)
  - vs. Topic: Different intent and processing order
- **Advanced Message Queueing Protocol (AMQP)** - P2P messaging protocol where client communicates with broker (e.g., RabbitMQ)
  - Messages published to **exchanges**, which distribute message copies to queues
  - Exchange types: Direct (Match), Fanout (All bounded queues), Topic (Wildcard match)
- **Persistent** - Messages stored until next node receives
- **Transient** - Messages only buffered for some time

08. Messaging Patterns

- Async. and enables enterprise integration
- Message contains: Header with message type, Payload

Message Channel

- **Return Address** - Tells replier where to send reply to
- **Correlation ID** - Specifies which request this reply is for
- **Request-Reply Chaining** - Chain using correlation IDs
- **Invalid Message Channel** - Handles erroneous messages
- **Dead Letter Channel** - For failed-to-deliver messages
- **Datatype Channel** - For specific type of data (RabbitMQ: Direct exchange chooses correct channel)
- **Pub-Sub Request-Response Pattern** - Sender communicates with multiple services via Pub-Sub Channel (Topic), but all responses aggregated back using queue

Message Routing

- **Simple Router** - Routes 1 channel to many channels
- **Composed Router** - Combination of routers
- **Context-Based Router** vs. **Content-Based Router**
- **Msg. Filter** - Content-based router with 1 output channel (e.g., Pub-Sub Channel with filters)
- **Msg. Splitter** - 1 message → Multiple messages
- **Msg. Aggregator** - Multiple correlated msgs. → 1 msg.
- **Message Scatter-Gather** - Broadcasts 1 message to services and aggregates replies into single message

Message Transformation

- **Msg. Translator** - Converts msg. format
- **Canonical Data Model** - Use common data format; requires 2 translators per service to translate in and out

Message Endpoint

- **Msg. Endpoint** - Interface bet. service and msg. system; channel-specific and distinct for send and receive
- **Polling Consumer** - Service controls when to consume
- **Event-Driven Consumer** - Consume on receive

09. Object Interaction Patterns

- **Design Pattern** - Solution to a problem in a context
  - Categories: Creational, Structural, Behavioral
- **Data Transfer Object (DTO)** - Carries data between processes to reduce number of remote calls

Structural Design Patterns

- **Bridge** - Split large class into separate hierarchies of abstraction and implementation (e.g., Shape with color)
- **Proxy** - Obj. sub. that controls access to original obj.
- **Adapter** - Allows objects with incompatible interfaces to collaborate (Similar: Microservices, Msg. translator)
- **Facade** - Provides simple unified interface to a set of subsystem interfaces (Similar: API Gateway)

Behavioral Design Patterns

- **Observer** - Subscription mechanism to notify observer objs. about events that happen to a subject obj.
  - Pull Model: Observer calls from subject when notified
  - Push Model: Subject pushes snapshot on state change
- **Mediator** - Forces objects to communicate via a mediator object (Similar: Event channel)

10. Guiding Principles

- **Modularity** - Independent modules that contains everything necessary to execute 1 functionality
- **Single Resp. Prin. (SRP)** - Limit module to 1 purpose
- **Sep. of Concerns** - Isolate distinct areas of functionality
- **Loose Coupling** - Module knows little about others
- **High Cohesion** - Bundle related behavior together
- **Program to Interfaces** - Write code that depends on abstractions, rather than implementations
- Information Hiding, Encapsulation