CS3219
AY24/25 Sem 1
github.com/jasonqiu212

01. Introduction

Software Types

- **Embedded** - Hardware system with software designed for performing specific set of functions
- **Real-time** - Timing is important
- **Concurrent** - Different computations run across the same or overlapping time periods
- **Distributed** - Runs across more than 1 computer, usually via a network
- **Edge Computing** - Computation done at leaf nodes
- **Cloud Computing** - Host software on ext. data center
  - **Cloud-enabled** - Legacy applications modified to run on the cloud (vs. cloud-native)

Software Development Process

- **Waterfall** - Sequential approach good for stable req.
- **Agile** - Iterative development with feedback loops and quick responses to changes
  - **Scrum** - Work done in sprints, where a subset of the product backlog is cleared

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

- **Requirement** - Capability needed by a user or must be met by a system

Types of Requirements

- **Business Req.** - Why the organization is implementing the system, e.g., reduce staff costs by 25%
- **User Req.** - Goals the user must be able to perform with the prodduct, e.g., check for flight on website
- **Functional Req. (FR)** - Specifies what a system does, e.g., website can export boarding pass
- **Business Rules** - Policies that define or constrain requirements, e.g., staff gets 40% discount
- **Quality Attributes** - How well the system performs, e.g., Mean time bet. failure $\geq$ 100 hours. A type of non-functional req.
- **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.
- Flow: Business Req. $\rightarrow$ **Vision and Scope Document** $\rightarrow$ User Req. $\rightarrow$ **User Req. Doc.** $\rightarrow$ FRs $\rightarrow$ SRS

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 behavaior 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
- **Change Control** - Process to ensure changes to a product are introduced in a controlled and coordinated way

Quality Attributes

- Different apps have different quality attributes
- 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** - Planned up time of system
  - Availability $= \frac{\text{Up time}}{\text{Up time+Down time}}$
- **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 accommodate more users, servers, locations, and etc.
  - Vertical Scaling: Add capability of machines
  - Horizontal Scaling: Add more machines
- **Efficiency** - How well app uses hardware, network, etc.
- **Modifiability** - How easily code can be understood, changed, and extended
- **Portability** - Effort needed to migrate software from 1 environment to another
- **Reusability** - Effort needed to convert software component for use in other apps
- **Verifiability** - How well software can be evaluated to demonstrate that it functions as expected
- Others: Maintainability, Testability

03. Software Architecture

- Contains components, connectors, config. (structure)
- **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
- **Message** - Data sent to specific address
- **Event** - Data emitted from component for anyone listening to consume
- **Decomposition** - Breaking down a system
  - **Horizontal Slicing** - Designing by layers
  - **Vertical Slicing** - Designing by features
  - Criteria: Modularity, coupling, cohesion

Architectural Styles

- Categories:
  - How is code divided? (Technical partitioning, domain partitioning)
  - How is system deployed? (Monolithic, distributed)
- **Layered** - Software organized as layers of components
- **Pipe and Filter** - Data flows through components (Data source, filters, data sink) via pipes
- **MVC** - Model (Business logic), View, Controller (Coordinates between view and model)
- **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
  - Not an architecture by itself
  - **Hypermedia** - Combo. of multimedia and hyperlinks
- 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 caheable 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 can be identified and manipulated by components (e.g., HTTP DELETE /user/:id)
- Code-on-demand: Optional; Allow client functionality by downloading executable code
- Pros: Less coupled, scalability
- Cons: Being stateless decreases network performance

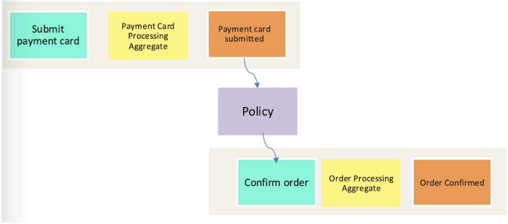
04. Microservices Architecture

- **Microservices App.** - App. as suite of small services
- Each microservice:
  - has **well-defined** (Cohesive) business capabilities: Boundaries align with business capabilities and features are closely related
  - developed/deployed independently
  - communicate with each other through well-defined mechanisms (Sync. or Async.)
- How do we identify boundaries of microservices?

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 contexts model interactions between sub-domains
- **Aggregate** - Cluster of related entities and value objects that are part of bounded context
  - 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

Event Storming



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

Data Patterns in Microservices

- Motivation: How do microservices manage data?
- **Database-server-per-service Pattern** - Each service has its own database server
  - 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
- **Data Lake Pattern** - Aggregate data from microservices into read-only, query-able data sinks
- **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)
  - For modifying data through multiple microservices
  - Better for steps that are harder to compensate at end
- **Event Sourcing** - Store stream of facts/events that got app. into current state, instead of storing current state
  - Different from relational/NoSQL
  - 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. commands write into Kafka queue of events; materialized view database derived from events
  - Pros: Single write model can add data into many read models, Can scale independently

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** - Code can be easily versioned
- **Orchestration** - Rely on central brain to drive processes
- **Choreography** - Inform each component of its job, and let it work out the details
- Service communication: Sync/async? 1-way or 2-way?
  - **Event-Driven Communication** - See EDA
  - **Request-Response Communication** - "Do this"; Generally sync. (Latent and 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. Event-Driven Architecture

- **Event** - Notifies services that something happened and/or copies data to other services
  - **Initiating Event** - From end user
  - **Derived Event** - Created due to initiating event
  - Structure: Key-value pair (**Unkeyed** - No key; **Entity** - Unique key; **Keyed** - Key not unique; For partitioning)
- **Event-Driven Architecture (EDA)** - **Event-based** with **async.** communication
  - **Real-time data** - Published as it is generated
  - Components: Producers, Brokers, Consumers
  - Can combine into event-driven microservices
- **Broker** - Receives events, stores events in queue/partitions, and provides events for consumption
  - Properties: Immutable, Ordering, Indexing, Partitioning, Infinite retention, Replayability
  - **Topic** - Category of queues/partitions
  - **Partition** - Indexed queue that **persists** after pop
  - Consumer consumes by index of last message it read
- Pros: Fast since async., Scalable broker, Less coupled

06. Scalability

- **Scalability** - System property to handle more work
- Scaling services in monolithic applications:
  - **Scale Up** - Upgrade server
  - **Scale Out** - Run multiple instances
    - **Load Balancer** - Chooses service replica to execute user request
    - **Session Store** - Stores user's session across replicas
- Scaling databases:
  - **Caching** - Good for data that is frequently read and rarely changes
  - **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.
- Split system into microservices to help scalability
- **Pod Archi.** - (Swim lanes) Place group of services/replicas inside boundary to contain failures

07. Asynchronous Communication

- Async pros: Responsive, Availability
- Cons: More complex error-handling