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 prooduct, 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 $= \frac{\text{Up time}}{\text{Up time}+\text{Down time}}$
- **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 accommodate more users, servers, locations, and etc.
  - Horizontal Scaling: Add more machines
  - Vertical Scaling: Add capability of 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

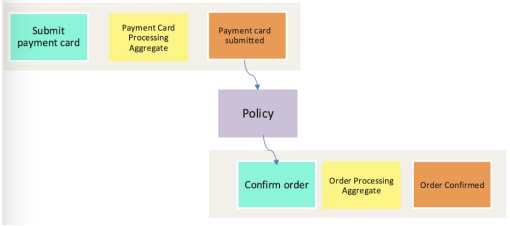
04. Microservices Architecture

- **Microservices App.** - App. as suite of small services
- Each microservice:
  - have **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