CS3219

AY23/24 Sem 1

github.com/SeekSaveServe

Types of software application

- **Embdedded**:Limited and specific set of functions. Found in industrial or consumer devices.
- **Real-Time**: Where timing is important. Hard — must be on time; firm — little room for flexibility; soft — greater room.
- **Concurrent**: Different computations run across the same or overlapping time periods
- **Distributed**: Runs across more than one computer. Usually connected via a network.
- **Edge Dominant**: Computation performed at end points instead of centrally.

Software Deployment

- **Orchestrator Vs Container**: Container provide the platform for building and dsitributing, orchestrator scales and fault tolerates and communicates among containers
- **Serverless**: More lean than containers, automatic scaling
- **Benefits of Containers**: Runtimewith code, dependency and change management, environment management, isolation & security

SDLC, Scrum, CI/CD Agile: emphasises iterative development and cross-functional collaboration. • Individuals and interactions over processes and tools.• Working software over comprehensive documentation• Customer collaboration over contract negotiation • Responding to change over following a plan. **SCRUM**: Agile framework. •Work done in sprints, where a subset of the product backlog is cleared. •There is often a daily 15 min SCRUM meeting **CICD**DevOps combine development and operations to reduce time between committing changes to production, while ensuring quality. • **C-Int**: Auto build, unit test, deploy to staging, and acceptance test. Allows teams to detect problems early. • **C-Del**: Same as above, except with manual deployment to production. Ensures that every good build is potentially ready for production release. • **C-Dep**: Same as above but with auto deployment to production. Automates the release of a good build

Software requirements process A requirement is a capability needed by a user or must be met by a system, i.e. what should be implemented. Can be of the system behaviour, property or attribute. • **Sources**: document, interview, questionnaires, event-response, prototyping, observations

• **SRS** contain System Reqs, Functional Reqs, External Interfaces, Quality Attributes and Constraints. It describes what the software must do • **Backlog** is a repository of work to be done and facilitates planning and prioritisation of work

• **Types of Requirements• Business Reqs**: Why the organisation is implementing the system, e.g. reduce staff costs by 25• **User Reqs**: Goals the user must be able to perform with the product, e.g. check for a flight using the website. • **Functional Reqs**: The behaviour the product will exhibit, e.g. passenger shall be able to print boarding passes. • **Quality Attributes**: How well the system performs, A type of non-functional reqs. •**System Reqs**: Hardware or software issues, e.g. the invoice system must share data with the purchase order system. Affects functional reqs. • **Data Reqs**: Describes data items or structures, e.g. product number is alphanumeric. • **External Interfaces**: Connections between your system and environment, e.g. must import files as CSV. Affects functional reqs. • **Constraints**: Limitations on design and implementation choices, e.g. must be backwards compatible. A type of non-functional reqs. • **Business Rules**: Actual policies or regulations. Constrains business, user and functional reqs. Generally, the flow is: Business Reqs → Vision & Scope Docs → User Reqs → User Reqs Specification → Functional Reqs.

• **Quality Attributes** (External) • **Availability**: Measure of the planned up time during which the system is fully operational. • **Installability**: How easy it is to install the system for the end-user. • **Integrity**: Preventing information loss and preserving data correctness. • **Interoperability**: How readily the system can exchange data and services with other software and hardware. • **Performance**: Responsiveness to user actions. May also compromise safety if e.g. a safety system responds poorly. • **Reliability**: Probability of the software executing without failure for a specific period of time. • **Robustness**: Degree to which a system performs when faced with invalid inputs, defects and attacks. • **Safety**: Prevents injury or damage to people or property. • **Security**: Authorisation, authentication, confidentiality, etc. • **Usability**: User-friendliness and ease of use. (Internal) • **Efficiency**: How well the system utilises the hardware, network etc. • **Modifiability**: How easily designs and code can be understood, changed and extended. • **Portability**: Effort needed to migrate the software from one environment to another. • **Reusability**: Effort required to convert a software component for use in other apps. • **Scalability**: Ability to grow to accommodate more users, servers, locations, etc. without compromising performance or correctness. • **Verifiability**: How well the software (components) can be evaluated to demonstrate that it functions as expected.

Prioritisation Techniques High-Med-Low. Must-Should-Could-Wish. $100 appro0ach.

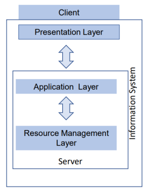
Validation vs Verification • **Validation** is about whether you have the right reqs, and if they trace back to business objectives. • **Verification** is whether you have written the reqs right, i.e. complete, correct, feasible, priority, unambiguous. Can be checked informally by passing the reqs around, or formally through formal inspection

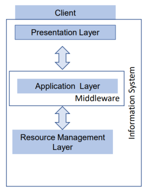
Building blocks of architecture Architecture Styles are recurring architecture designs. They describe structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. • **Component**: Models application-specific function • **Connector**: Models interactions between components for transfer of control and/or data • **Configuration**: Topology or structure

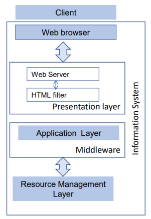
Message and Event Message is some data sent to a specific address, Event is some data emitted from a component for listening components to consume. Events are ordered in sequence of creation and immutable

Decomposition & Modularity • **Horizontal slicing**: design by layers (controller, service, repo) • **Vertical slicing**: design by feature (order, user, policy) • **Ports & Adapters**: Keep domain separate from technical details •**Modularity** Shorter dev time, better flexibility and comprehensibility **Cohesion**: Type, functional, layer, communicational, sequential, procedural, temporal, utility **Coupling**: Content, common, control,data, external, temporal, inclusion import

Layered Architecture Hierarchy with layers providing service to the layer above it and acting as a client to the layer below it. Layers contain various elements, and communicate via procedure calls (connector). **Pros** Abstraction. Partitions complex problems. Easy to substitute one layer for another. Easy to reuse and port. **Cons** Hard to "layerise" some systems, esp. when high-level functions needs coupling with low-level implementations. Performance degradation due to communication between layers.

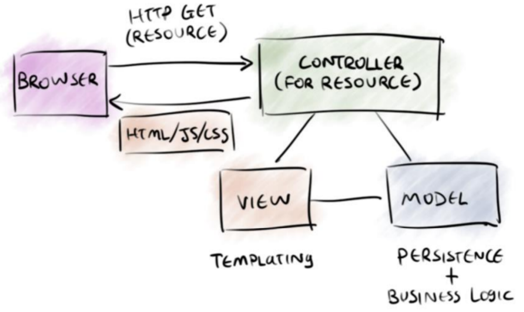






Pipe and Filter Filters are stateless stream transducers that incrementally transform data **Pipes** are stateless and simply transfer data. **Constraints** Independence; i.e. filters should not share state. **Anonymity** i.e. filters should not know which filter is before or after them. **Concurrency** i.e. should be able to run the filters concurrently. **Pros** No complex component interaction. Easy to reuse. Easy to parallelise. Easy to maintain and enhance. Good for image processing. **Cons** Not good at interactive applications. May need a common representation, i.e. pack and unpack costs.

MVC MVC provides SOC between data storage, data handling, data visualisation, user interaction through modularity. It also provides higher extensibility and testability. • **Model**: Contains core functionality and data. Updates the view when it changes. • **View**: Display info to user • **Controller**: Handle user input and pass info from view to model • **Web MVC**: Controller handles HTTP requests, model selection, view preparation, and the additional responsiblity of mapping requests to handler



SPA JS program downloaded int othe browser and runs continuosuly. Send query and retrievedata without refreshing the page. **Pros** Fuid experience, save bandwidth, reduce perceived latency **Cons** vulnerable to XSS attacks **Webb MVC - SPA** MVC is in browser, and SPA makes HTTP API requests against a set of resources served by API Controllers that replies with JSON

REST Basics Defines constraints for transferring, accessing and manipulating textual data repr (of hypermedia) in a stateless manner to provide uniform interoperability between different web apps. **Pros** Less tightly coupled systems, scalable, usable, accessible. **Cons** Stateless may decrease network performance bby increasing repetitive data sent in series of queries, may degrade efficiency due to uniform interface.

REST Constraints • Client-Server: Client not concerned with data storage, server not concerned with user interface or user state.Separation of concerns. Client and server can evolve independently. **Stateless** Server-client interactions self contained, full information as part of query params, headers or URI. Improves scalability (server can free up resource), reliability (recover from partial failures), monitoring (analyze request through request data) **Cacheable** response include if data is cacheablea nd for how long, improves network efficiency but can potentially retrieve stale data. **Layered System** Application organised as a layered system with each layer not seeing beyodn the immediate neighbor. Restricts complexity to individual layer, enables data transformation, allows load-balancing and caching at intermediary layers. **Uniform Interface** Uniform way of interacting with a srver that is decoupled from hardware and efficient for large-grained hypermedia data transfer. Components understand resource by inspecting its representation.

```
HTTP  GET  /device−management/managed−devices/{i
```

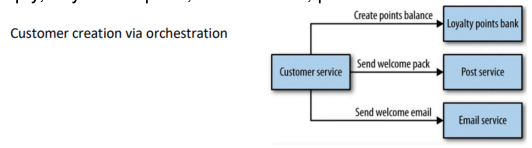
Code-on-demand Simplifies the client from having to pre-implement all functionality, allows extensibility.

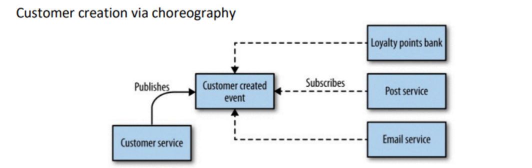
Microservice Definition A microservice is an independent, standalone capability that communicates through lightweight communication. **Organised around business capabilities** as these capapbilities invariantly need transfer of info. **Loose Coupling**, components should not depend on each other's implementation and their interaction in the real domain **Owned by Small teams** loosely coupled organisations create more modular, less coupled systems **Independently deployable** each microservice has their own deployment, resources, scaling and monitoring requirements.

Domain A domain model represents a view of the problem domain and a critical and fundational concept behind business needs. **Subdomains - SOC**, which can be classified as Generic(facilitates the business and is found across multiple domains e.g. invoicing / communication. Can be off the shelf) Supporting (auxillary but not a key differentiator) Core(key differntiator of business)

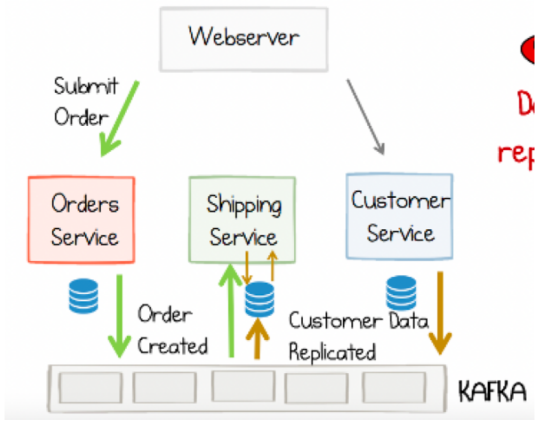
Domain Driven Design domain model should be isolated from technical complexities **Ubiquitous Language - increase cohesion** A language structured around the domain model and used by all team members to connect all activities of the team with the software. **Bounded Context-SOC, single responsiblity** Separates ubiquitous languages. As we have many problems, we may not be able to use the same language throughout. But we can cluster sets of problems where the same specialised

language can be used, and make boundaries. Contains code for a single subdomain. **Aggregate** cluster of related objects that is treated as one unit for data changes. Changes to the aggregate will either all succeed or none will succeed. Aggregate's state can only be modified using its public interface (aggregate root entity)
Identifying Microservice Small set of responsibilities, apply SRP. Services should encompass entire bounded context. Break services around aggregate boundarie.
More about Microservices Database Each service has its own DB, duplication of data. **Orchestration** rely on a central brain to guide the process. **Choreography** inform each part of the system of its job **Client side discovery pattern** Client queries a service registry and determines the network location of available service instances and decides which one to query based on load balancer. **Server-side discovery pattern** client makes a request to a service via load balancer, load balancer queries the service registry and route requests **Service registry pattern** data base of service instances and locations, client or router query the registry to find available instances **Deployment** service instance per host or service instance per container **Service Routing Collaboration** API Gateway, Orchestration, Choreography **Service Discovery**, client side, server side, service registry **Service Communication** sync - request reply, async - request, event-driven, pub-sub

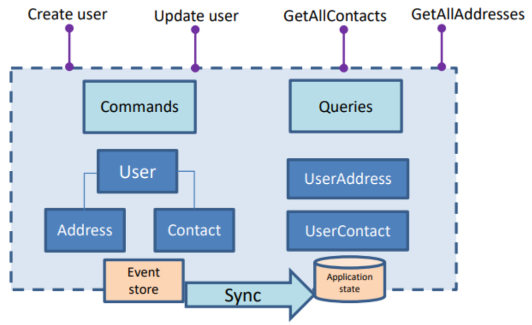




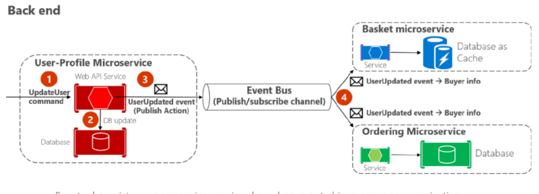
Event-definition events are anything that happen within the scope of the business communication structure and acts as both data and means of async communication. **Unkeyed events** describe a singular statement **Entity** is unique and provides continual history of the tate of an entity **Keyed** Contains key but does not represent entity, used for partitioning stream of events to gurantee data locality **Event Driven Architecture** is where decoupled applications can asynchronously communicate by producing and consuming events via an event broken (facilitates loose coupling) **Event Broker (kafka)** receives data and publishes it on source-specific topics **Producers** generate stream of events and **Producers** listen for events. Producer does not know hich consumers are listening **Advantages over request driven architecture** detach creation and control of data from consumption and decouple systems so they can be independently scaled and updated, handle high volumes of data with low latency, support real-time processing and analytics, be more scalable and resilient to failures, does not follow synchronous pattern. Request driven is better for web APIs, databse queries where a client needs specific infroamation, whereas event driven is better suited where components need to react to changes in state (e.g. IoT, real-time analytics, etc) **Advantages of Event Driven Microservices** Granularity (easy to rewrite), scalability, technological flexibility (different stacks), business requirement flexibility, Cont Del support (easy to ship), high testability **Event brokers** rabbitMQ (50k messages per second, persistent and supports one-to-one and many to many), kafka (1 mil messages per second, persistent but only supports one-to-many), reddis (1 mil messages per second, not persistent, both one-to-one and many to many)**Event driven frameworks** spring cloud stream



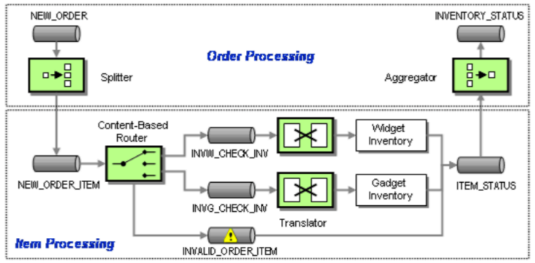
Event Sourcing A data storage pattern where the events are stored in an append-only log.Event log provides an audit of the system activities and allows current state to be recreated by rewinding the log in order.
Command Query responsiblity Separation pattern CQRS separates write path - command interface from read path - query interface (SRP, SoC, ISP). If anything goes wrong, internal state of DB can be recovered from log, and writes and reads can be optimised independently. Writes enter kafka on the command side, and event stream can be transformed in a way that suits the query using KSQL **Benefits** a single write can be read by many reads



Synch communication caller/sender waits for response from sender/caller. Msg processed only once
Remote Procedure Call (RPC) allows a program to execute a procedure in another address space (distributed machine) that mimics serial thread execution. Client calls the client stub (local call) and parameters are packed into a message (marshalling) which is sent to the server machine. The server stub unpacks the parameters (unmarshalling).
Asynch communication (AMQP) does not wait for response and continues executing whatever code is necessary, enables independent functioning of sender and receiver and one-to-many communication
HTTP/S, asynchrony in request-reply pattern client sends request for a resource to a server and the server sends back a response. This decouples backend from the requester and allows for a response to be sent when the backend is async. 1. (client) Call API 2. Enqueue message 3. (server) Reply with status end point 4. Receive message 5. Process message write result to blob $\longrightarrow$ [Storage Blob] 7. $\longleftarrow$ check for blob 6. $\longleftarrow$ (client) Poll status end point
Advanced Message Queueing Protocol (Rabbit MQ) AMQP is a peer-to-peer messaging protocol that enables conforming client applications to communicate with conforming messaging brokers. Messages are published to exchanges that distribute message copies to queues based on messsaging attributes. Then brokers deliver message to consumers or consumers fetch messages from queues on demand. **Exchange types** Direct: Message is routed to the queues with matching binding keys. Fanout: routes messages to all of the queues bonund to it. Topic: The topic exchange does a wildcard match between the routing key and the routing pattern specified in the routing.
Multireceiver message can be published to topics which serves as a broadcasting station for subscribers.
Pub/Sub pub of data does not specifically direct to a sub, instead pub classifies the message the the subscriber subscribes to receive messages it is interested in (from broker) **Kafka** each kafka server or broken is responsible for delivering the message to the right consumer, more can be added without downtime which enables horizontal scaling. Messages are grouped by topics. Within each topic, msgs are partitioned by their key. Msgs are added in append only fashion and read in FIFO order. One consumer group works to consume a topic, with each partition only consumed by 1 member, allows hoirzontal scaling of consumers.


Back end
User-Profile Microservice
1
UpdateUser command
Web API Service
3
UserUpdated event (Publish Action)
2
DB update
Database
Event Bus (Publish/subscribe channel)
4
UserUpdated event → Buyer info
UserUpdated event → Buyer info
Basket microservice
Service
Database as Cache
Ordering Microservice
Service
Database
Eventual consistency across microservices based on event-driven async communication

Message Intent Command message specifies a function or method to invoke on the receiver. Document message transmits its data structures to receiver. Event message notifies the receiver of a change in the sender (just notification, no action specified). **Message Channels** Connect collaborating senders and receivers, has one way and two way. **Return Address**: Request contains an address to tell the replier where to send reply to.
Correlation ID: Specifies which request the reply is for. Can be chained (allows tacing of message path). **Message Sequence**: Basically break the message down into smaller segments. **Point to Point (P2P)**: Request processed by single consumer. **Publish-Subscribe Channel**: Request broadcasted to all interested parties via topics. **Invalid Messages**: Queue to move a message to when it cannot be interpreted. **Dead Letter**: Queue to move a message to when it cannot be delivered. **Datatype Channel**: Separate channel for each type of data, e.g. XML, byte array, etc.
Message Routing routers consume messages from one message channel and reinsert them into different message channels depending on conditions. **Context based router** examines the message content and routes based on data contained, needs knowledge of all possible reciepients, a change in recipient list changes the router. Uses **Message Filter** which has only a single output channel. **Message splitter** splits 1 msg into many msgs. **Message Aggregator** combines many correlated msgs into 1 msg.



Message Scatter-Gather broadcasts 1 msg to a number of consumers concurrently and aggregates the replies into a single message (good for requesting requests from multiple services) **Message Translator** converts messages from one form to another. Data can be first translated into a **Canonical Data Model** before being translated into the target format to reduce number of translators needed.
Message Endpoints Interface between app and messaging system. Channel-specific, one instance handles either send or receive. Can be synchronous and proactive, i.e. polling consumer, or can be asynchronous and reactive, i.e. event-driven receiver
Modular Monolith loosely coupled, highly cohesive modules. **Pros** independent modules, easier to maintain, test, develop, reuse. Each module knows as little as it needs to know about other modules (relate behaviours tgt by boundaries, limit the number of different types of calls)
Code Smells Duplicate code, large class, lazy class.
Design Smells a deeper problem at the respective abstraction level (e.g. large classes with a lot of responsibilities). **Problem to interface** rely on interface (abstract class) to decouple from implementation.
Composition over inheritance since inheritance has a tight coupling between class and subclass. **Encapsulate what varies** the component that changes frequently should be encapsulated (e.g. into a factory class).
Design Pattern is a soln to a problem in a recurring context. **Creational** encapsulate details about the acutal creation (factory, singleton, builder), **Structural** define relationships between classes to form larger objects (adapter, decorator, facade), **Behavioural** define manners of communication between classes and objects (observer, strategy, template method)