- Get feedback from somebody by explain inng it
- Short and intense practice

==========

Microservice Architecture Patterns

- Layered architecture
- Hexagonal architeacture
- Microservice architecture
- Microservice chasis: provides basic template for microservice architecture example spring boot

1. **Decomposing Strategies**:

- System requirements and user stories
- User stories (when + action + tuen)
- Define domain objects from user stories (All noun parts of user stories)

- ○ Define system operations from user stories (All verb parts of user stories)
- ○ Defining different services by applying decompose by business capabalities pattern (identify different business requirements and the corresponding grouped services)
- ○ Defining different services by applying decompose by sub-domain patterns (identify different domain, related sub-domains and tje corresponding grouped services)
- ○ Also apply some of the object oriented design principles while defining services like Single Responsinilty Principle, Open Closure Principle, Common Closure Principle
- ○ Then treat each services as a separate sub-domain with its own

domain model which could eliminate presence of God classes

- ≡ ○ Once each services are identified define each services set of service endpoints (operations + events)
- ≡ ○ Some of these service operations will be consumed by external clients and others are by other services for collobrations
- ≡ ○ Services may publish events to enable collabrations with other microservices or to notify events informations to external clients
- ≡ ○ Next assign each of the previously defined system operations with service API´s.
- ≡ ○ Finally decide how the services collabrate in order to handle each prexondition or precondition of each system operations

Domain models, system operations,

services and service collabarations.
Domain driven design and using bounded context

2. Interprocess communications:
Communication Styles

- Synchronous communication between client and server or between services (HTTP based REST and gRPC)
- Asynchronous communication between client and server (AMQP and STOMP)

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

- Communication message formats can be either text based formats such as JSON/XML or binary format such as Avro or Google Protocol Buffer (ProtoBuf)

Synchronous REST

- API first design is essential, so that there will be clear contract between server side and client side bedore starting code implementatio. API`s need to have versions like v1,v2 etc so that new functionality can be provided in api with backward compatability

Note: in case of http rest api design, enabling the client to retrieve multiple related objects in a single request will be a problem. For example client want to retrieve an order and the order's customer. For this we can use an external plugin called Graphql which is a query language tool for our api.

- Handle partial failure of services using the Netflix Circuit breaker pattern
- Implement application provided service discovery using netflix Eureka or use the deployment

infrastructure like kubernetes provided service discovery (through DNS names and Virtual IP)

Asynchronous Messaging

- Event based: Usually one service ll communicate with other in a asynchronous way by using any message brokers like rabbitmq or kafka.

For synchronous communication; Can use FeignClient to call another service aling with circuit breaker and retry logic

3. API based microservice architect

- REST architectural style
- Facade design pattern
- Proxy design pattern
- Stateless servuce pattern

4. How to compose microservices together

≡ ○ Broker composition pattern
≡ ○ Aggregate composition pattern
≡ ○ Chained composition pattern
≡ ○ Proxy composition pattern
≡ ○ Batch composition pattern

5. Data consistency across microservices

≡ ○ Two phase commit
≡ ○ Saga pattern
≡ ○ Saga choreography and orchestration
≡ ○ Eventual consistency

6. Centralize accesa to microservuce using an API Gateway

≡ ○ Centralized access by api gateway
≡ ○ Can implement common functionalities like authentication authorization logging etc
≡ ○ API composition pattern:

responsible for calling multiple services and aggregate the data. Can be run behind an API Gateway

- ○ Cache can be used either at the api gateway level or at services level, to return the previously stored data when the same request requested. Spring boot cache along with cafine can be used
- ○ Can use enterprise level cache system like Reddis for caching purpose
- ○ Spring cloud gateway
- ○ Kong api gateway
- ○ API Gateway can use Spring Cloud OAuth2 Authorization server for authenticating and receiving jwt tokens
- ○ Access token and refresh tokens
- ○ OAuth2: resource server, authorization server, resource

owner and client

- ○ JWT tokens: header, payload and signature. Header contains info about what type of message and signature hash code. Payload contains user id name and list of access role informations, signature contains encoded header, payload info along with some secret
- ○ If access token is expired we will extend the token expiry time by using refresh token. Each servuce then gets the access token, extracts it to get the user details usinh the same secret

7. Split monolithic databases across microservices

- ○ Microdatabases
- ○ Event driven
- ○ Event sourcing and event store: Axon framework can be used to

save the message as series of events. So in the DB we will store snapshot of the data every time, so that entire history of the data available and which can be replied any time

- Transaction output tables: services take the data from listeners do some calculation on it, push the result in the DB and also push the same message to broker channel. In the same trabsaction it writes the message to transaction output tables also for CQRS purpose.
- Transaction log tailing: tools like **Debizium** which can tail the logs from transaction output table and publish to broker channel. So the service at the query side will listen and save such message into separate DB for query purpose

- ○ Polling publisher: using some batch jobs which repeatedly take the data from transaction output tables and publish it to message broker channels
- ○ CQRS: Command Query Responsibility Separate. Separate services for command operations like create update delete and query operation like fetch
- ○ Greenfield database approach
- ○ Brownfield migration strategy

8. Make microservices more resilient

- ○ Timeout design oattern
- ○ Circuit breaker design pattern (Hystrix or Resilience4j)
- ○ Retry design pattern
- ○ Bulkhead design pattern

9. Make microservices backward compatibility

≡ ○ Versioning strategies

10. Define and document microservice contracts

≡ ○ Consumer driven contract
≡ ○ Resource based microservice
≡ ○ Action based microservice
≡ ○ Task based microservice
≡ ○ Interface definition language
≡ ○ Swagger
≡ ○ Spring Rest Docs

11. Microservices centralized logging

≡ ○ Spring boot logback configurations
≡ ○ Logstash or FluentD
≡ ○ Elastic search
≡ ○ Kibana

12. Provide reporting from distributed microservice data

≡ ○ Reporting service calls
≡ ○ Data push applications
≡ ○ Reporting event subscribers
≡ ○ Reporting event via gateway
≡ ○ ETL and data warehouses

13. Automate on premises microservices

≡ ○ Continous intehration
≡ ○ Continuous deployment
≡ ○ Continuius delivery
≡ ○ Azure devops or Jenkins

15. Cloud based microservice infraatructure

≡ ○ On premise
≡ ○ IAAS
≡ ○ PAAS
≡ ○ SAAS
≡ ○ Hybrid Approach

16. Microservices configuration

- ○ Deployment servers
- ○ Externalized configuration pattern
- ○ Configuration management tools
- ○ Containers
- ○ Spring cloud config with git
- ○ K8s ConfigMaps and Secrets
- ○ External dababase like Vault for storing sensitive data
- ○ Consul for configuration management

17. Microservices registration and discovery

- ○ Client side discovery: using spring cloud kubernetes with Ribbon Clinet and RBAC
- ○ Server side discovery: k8s by default provides this through DNS
- ○ Service registartion
- ○ Spring cloud Eureka or **Consul** for service registry and discovery

18. Monitoring microservices

- Monitoring key metrics
- Monitoring SLA metrics
- Minitoring dashboards
- Alerting and monitoring
- Defining threshold for alerts
- Monitoring tools and patterns
- Spring boot actuator provides multiple metric details by default like health check details etc
- Spring boot, MicroMeter, Prometheous and Grafana can be used for alert and monitoring purpose
- Distributed tracing patterns by using spring cloud sleuth and zipkin which provides unique correlation id across multiple services for each request
- Exceprion or error tracking : tools like Sentry can provide exveption

monitoring support

Other common tools:

- Batch jobs(Spring batch jobs)
- Jobs scheduler(Spring scheduler, Quartz Scheduler)
- Logging all environment variables while server startups
- Memory and CPU usage analysis by using VisualVM
- Load test or performance test through JMeter
- Code quality checkin through SonarQube
- Maven or Gradle for build automation
- Doxker for running application
- Java testing through jUnit and Mockito
- Findbugs: for checking bugs in the project
- Code coverage reporting tool:

Cobertura

- Java decompiler
-

Hexagonal Architecture

- Inbound adapters and ports, outbound adapters and ports
- Layer 1: server(external client will call these rest apis defined in this layer) request and response objects, client(application uses this layer to make a call to another internal service or external service)
- Layer 2: Command and Query. Each laywr will have its own service, action and entity classes.
- Layer 3: spring data repository to accesa databases
- Layer 4: message broker with publisher, listener and actuall message itself

- Inbound adapters: server or listener
- Inbound ports: services and actions
- Outbound adapters: client or publisher
- Outbound ports: data repository
- Domain objects: command entity and query entity

==========

Java 8
Functional and Reactive Programming

Functional Programming

Functional programming paradigm is built upon the idea that everything is a pure function.
Java can uses functions as high class citizen.
Functional interface and lambda