17/9/19

Micro Services

MS is a SOA pattern wherein appl^n are built as a collection of various small & independent service units.

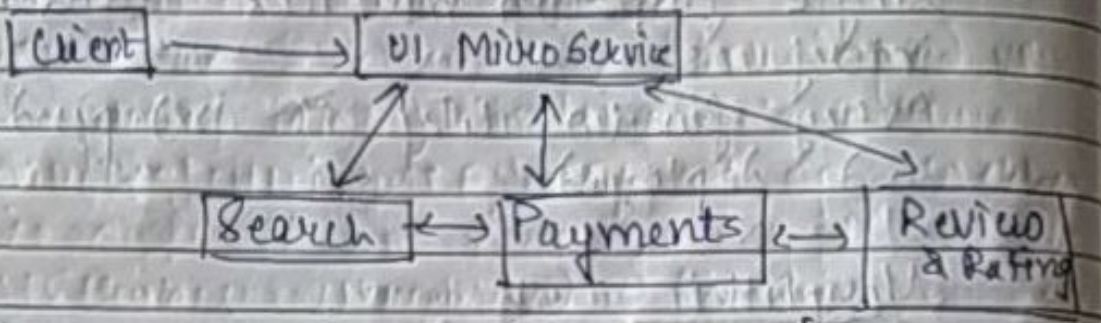
Single-function modules with well-defined interfaces.
Every functionality is independently developed and deployed.

Monolithic arch.
Service accessed as single entity
code for various modules (Search / Payment) lies on the same server.
The entire service needs to be deployed again even when a small change is made in a single module.
It may happen that at certain time, a module is accessed more than other modules. If developed in MS, we only need to scale the particular module of business logic; whereas monolithic arch. demands scaling of entire appl^n
Only single instance of a service is possible under mono. arch.

→ Developers of traditional WS do not have any info. about who is going to use the service. Whatever fits into the standard def. (WSDL) can use the service.
MS developers are well aware about

who is going to use the service being developed.

→ MS provides tech. heterogeneity. Service can be deployed on different servers as well



Response generated at runtime, i.e entire HTML page can be rendered at runtime.

MS	Monolithic
* Every unit of entire appln should be the smallest, and should deliver one specific business goal.	* A single code base for all business goals.
* Service startup relatively quick	* Service startup takes more time
* Fault isolation is easy	* Fault isolation is difficult. If any feature goes down, the complete sys. goes down. The entire appln needs to be rebuild, re-tested & re-deployed.

* Can be loosely coupled.	* Should be tightly coupled.
* Businesses can deploy more resources to services that are generating higher ROI	* Individual resource allocation not possible
* More hardware resource could be allocated to the service that is freq. used.	* Appln. scaling is challenging
* Data can be distributed (federated)	* Data is centralized
* Consistent & always available.	
* Small focused team. faster development.	* Large team & team management effort req
* Change in data model of one MS does not affect others.	* Change in data model affects the entire DB.
* Interacts with other MS by using well-defined interfaces.	* Not applicable.
* Focus on products, not projects.	* Focus on entire project.

* No more dependency b/w code bases
* Depends on single code base.

MS Challenges

i) MS rely on each other, and they need to comm. with each other.
ii) More serv. to monitor (developed using diff. prog lang.)
iii) MS arch brings plenty of op. overhead
iv) It is difficult to manage appl^n when new services are added to it.
OS update in phone → Mono. arch
Appl^n update → (Service update) MS arch
v) Skilled professionals req.
vii) Development is costly.

MS Advantages

a) Small in size
b) Focused (Module should function properly)
c) Autonomous
d) Technology heterogeneity
e) Resilience (Isolation of service)
f) Easy to deploy

SOA v/s. MS.

* SOA services are maintained in the org. by a registry which acts as a directory listing. Appl^n. need to look up the serv. in the dir.

* SOA ≈ Orchestration
MS ≈ Choreography

SOA	MS
* Biz comp. are exposed as Serv to outer world	* MS is part of SOA
* Business modules are dependent	* Independent
* Deployment is more time-consuming	* Less time consuming
* More cost-effective	* Less cost-effative
* Less scalable (relatively)	* Highly scalable
* Busi. logic comp. are stored inside single Serv domain	* Busi. logic can live across various domains

MS Tools

1) wiremock : Testing Microservice
2) Docker
3) Hystrix
4) Spring Boot

25/9/19 Ultimate version - Spring dependencies are included

⇒ start.spring.io
⇒ Project : Maven Proj
Lang : Java
Group : in.microservices
Artifact : movie-catalog-service

Options: Packaging - Jar
Java - 11
Search dependencies to add: Spring Web
→ Click Generate

// Similarly for movie-info-service and rating-data-service
⇒ Open project in IDE

Right click on package → New → Java class
Name: resources.MovieCatalogService

MovieCatalogService.java
@RestController
@RequestMapping("/catalog")
public class MovieCatalogService
{
—*— catalogItem ⇒ in package models
public List<> getCatalog(String movieId)
{
CatalogItem ci = new CatalogItem
(movieName: "super 30",
desc "education",
rating "4");
return Collection.singletonList(ci);
}
}

—*— @RequestMapping("/{movieId}")
... getCatalog(@PathVariable("movieId")
String movieId)

```
public class CatalogItem
{
    private String movieName;
    private String desc;
    private int rating;

    // getter setter methods
    // constructor
}
```

→ URL : localhost:8080/catalog/1
Output in JSON format

⇒ Open project movie-info-service
New Java class : resources.MovieResource
@RestController @RequestMapping("/movies")

```
public class MovieResource
{
    @RequestMapping("/{movieId}")
    public Movie getMovieInfo(@PathVariable
            ("movieId") String
                movieId)
    {
        return new Movie(movieId,
                movieName "Super 30");
    }
}
```

```
public class Movie
{
    private String movieId;
    private String movieName;

    //getter-setter methods & constructor
}
```

(R.C -> Generate)

Page No.
Date

Project View → Main → resource → application.properties
Add line:
server.port = 8082

26/9/19 In catalog service:

//copy Rating class or write that code again

public ... getCatalog (...)
{
List<Rating> ratings = Arrays.asList(
new Rating(movieId: "1234", rating: 4)
new Rating(movieId: "5678", rating: 3)

— ★ —

return ratings.stream().map(rating → { new CatalogItem(movieName: "Super 30", desc: "Education", rating: 4) }).collect(Collectors.toList());

}

— ★ —

RestTemplate rt = new RestTemplate();
Movie m = rt template.getForObject(url: "http://localhost:8085/movies/" + rating.getMovieId, Movie.class
return new CatalogItem(m.getMovieName(), desc: "Education", rating.getRating())

30/9/19 **JMS** (Java Message Service)
It is an API which supports the formal comm. called messaging b/w comp. in a network

- Provides a common interface for standard msg protocols.
- Used for Java applications only.
- Can send messages even when two machines do not know about each other and lie in two different continents. Serialization of objects is a must.

Benefits

1) Asynchronous
JMS is async by default. Msg arrives automatically to client as they are available

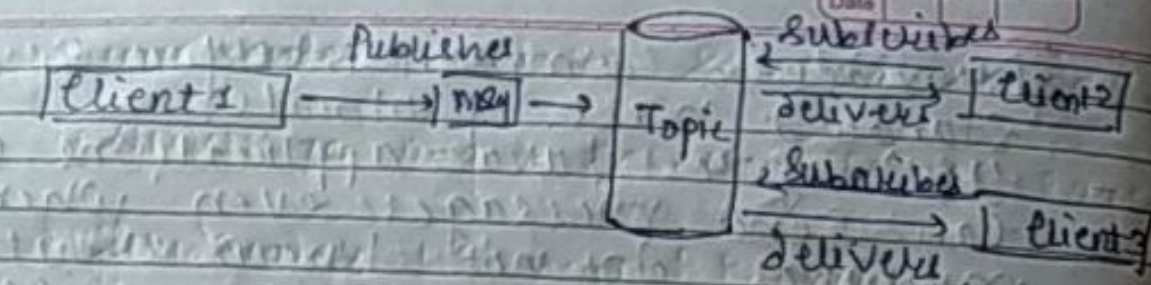
2) Reliable.
Assurance that msg is delievered once & only once.

→ Befor JMS, most msg products support either point-to-point or Publish/Subscribe approach.

JMS working

- Point-to-point msg domain
Built on basis of queues, senders & receivers. Every msg is addressed to a particular queue. Queues retain msg until they are consumed or expired.

- Publish/Subscribe Domain
One sender and multiple receivers.



JMS basic building blocks:

1) Administered Obj (Stored in JNDI Directory)
2) Destination
3) Connection (virtual con. with JMS Provider)
4) Session
5) Msg. Consumer (created by session)
6) Msg. Producer (created by session)
7) Message Listeners (Asyn. event handler for msgs.)

10/19

Steps:

1) Start GlassFish Server
2) Open admin console
(Right click on Server)
3) Resources → JMS Resources →
Connection Factories
New
Give JNDI Name
Resource Type : javax.jms.ConnectionFactory

Destination Resource
New
Give JNDI Name (myQueue)
Physical Dest. Name (abc)
Resource Type : javax.jms.Queue

4) On NetBeans, create new project
Java → ~~Java Appl^n~~ Web Appl^n
Name (Demo1MS)
(No Main class)

-Create new Java Package (Demo)

Libraries → Add Library →
~~Java EE API~~
Java EE Web 7
Java EE from Glassfish
Java EE Endorsed Web 7

Add new Java class (MySender)

```
public class MySender
{
    p. s. v. main (String[] args)
    {
        try
        {
            // JMSContext ctx = new JMSContext()
            InitialContext ctx = new
QueueConnectionFactory cf = (QueueConnectionFactory)
                ctx.lookup("myQueue-
                ConnectionFactory");
QueueConnection con = cf.createQueueConnection();
con.start();

QueueSession ses = con.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);

Queue t = (Queue) ctx.lookup("myQueue");
```

```
QueueSender sender = ses.createSender(t);
TextMessage msg = ses.createTextMessage();
BufferedReader b = new BR(new
                    InputStreamReader(System.
while (true)
{
   Sys.o.p("Enter msg :");
   String s = b.readLine();
   if (s.equals("end"))
   {
      break;
   }
   msg.setText(s);
   sender.send(msg);
   S.o.p("Msg sent successfully");
  } con.close();
}

public class MyReceiver
{
   p.s.v. main(...)
   {
      InitialContext
         :
         :
      con.start();
      QueueSession ses = con.createQueueSession
                  (false, Session.AUTO_ACKNOWLEDGE
```

```
Queue t = (Queue) ctx.lookup("myQueue");
QueueReceiver receiver = ses.createReceiver(t);
MsgListener listener = new
receiver.setMessageListener(listener);
s.o.p("Receiver ready")
s.o.p("Ctrl+c to shut down");
while(true) {
Thread.sleep(1000);
} }
}

public class MsgListener implements
                    MessageListener
{
  public void onMessage(Message message)
  {
    try
    {
      TextMessage textMes = (TextMessage)
                            message;
      if (textMes != null)
          s.o.p(textMes.getText());
    } catch(—)
      { }

}
```