**Microservices with Spring Boot and Spring Cloud**

**Web services:-**

* Providing service over the web.
* 3 Keys

1. Designed for **machine-machine** (application-application) interaction.
2. Should be **interoperable** - platform independent.
3. Should allow **communication** over the internet.

A picture containing text, screenshot, font, rectangle

Description automatically generated

A picture containing text, screenshot, font, line

Description automatically generated

* Data Exchange

A picture containing text, font, line, screenshot

Description automatically generated

* Request and Response Data Exchange Formats to make the service platform independent.
  + XML
  + JSON
* Every Web Service offers Service Definition

A diagram of a service structure

Description automatically generated with low confidence

* **Key Terminology in Web Services**
  + Request
  + Response
  + Message Exchange Format (XML, JSON)
  + Service Provider - Server (Web Service – which hosts the service)
  + Service Consumer - Client (Application – which consumes the service)
  + Service Definition – Contract b/w server and client
  + Transport – How the service is exposed? (HTTP – communication over web, MQ – communication over queue)

A picture containing text, design

Description automatically generated

* Web Services Groups/ Types:-
  + SOAP
    - **S**imple **O**bject **A**ccess **P**rotocol.
    - Pose restrictions on the exchange format.
    - No restrictions on Transport.
    - Service Definition used is WSDL (Web Service Definition Language).
    - XML – Request exchange format.
  + REST
    - **RE**presentational **S**tate **T**ransfer
    - Key is to make the best use of HTTP.
    - No restriction on Data Exchange Format 🡪 JSON is widely used.
    - Transport is always HTTP.
    - No Service Definition is available in REST 🡪 WADL, Swagger etc.,
    - REST 🡪 HTTP 🡪HTTP methods (GET, POST, PUT) 🡪 HTTP Status codes (200, 404 etc.,)
    - Key Abstraction 🡪 **RESOURCE**
      * Resource is an URI 🡪 /user/ranga/todos/1
      * Representations 🡪 XML, HTML, JSON

**Spring Boot:-**

* World before Spring Boot
  + Pom.xml 🡪 Dependency Management (REST APIs, Unit Tests etc.,)
  + Web.xml 🡪 (Dispatcher Servlet)
  + Context.xml 🡪 Spring Configuration (Component Scan, View Resolver etc.,)
  + Non-Functional Requirements ( Logging, Error Handling, Monitoring etc.,)
* Goal
  + Build **production-ready** applications **quickly**.
  + For Quickness,
    - Spring Initializer
    - Spring Boot Starter Projects
    - Spring Bot Auto Configuration 🡪 maven dependency jar
    - Spring Boot Dev Tools
  + For Production Ready,
    - **Logging**
      * Trace
      * Debug
      * Info
      * Warning
      * Error
      * Off
    - Different configurations for different environments (Profiles, Configuration properties)
    - Monitoring (Spring Boot Actuator)
      * Beans
      * Health
      * Metrics
      * Mappings

**Spring Boot Data JPA:-**

* Spring Data JDBC, Spring JPA, Spring Data JPA, Spring Hibernate
* CommandLineRunner which is executed at the tomcat startup
* Hibernate vs JPA
  + JPA defines specification. It’s an API.
    - Define entities.
    - Map attributes.
    - Manage entities.
  + Hibernate is the most popular implementation of JPA

**Functional Programming:-**

* Paradigm shift
* Stream
* Map
* Filter
* Optional 🡪 to deal with NullPointerException

**Spring Boot Microservices:-**

* Dispatcher Servlet 🡪 Front Controller Pattern [“/”]
* The above servlet is configured using Spring Boot Auto Configuration 🡪 DispatcherServletAutoConfiguration
* Return of json response 🡪 @ResponseBody + JacksonHttpMessageConverters
* Error Mapping 🡪 ErrorMvcAutoConfiguration
* Availability of Jars 🡪 starter projects Spring Web dependency

**Social Media Application**

Users REST API

Get users GET /users

Create users POST /users/{id} /user/1

Delete the user DELETE /users/{id} /user/1

Posts REST API

Retrieve posts GET /users/{id}/posts

Retrieve details of post GET /users/{id}/posts/{post\_id}

Content Negotiation

* Different Content Type 🡪 XML or JSON type etc.,
* Different Language 🡪 Dutch, Telugu etc.,
* Accept Headers 🡪 MIME Types: application/xml or application/json
* Accept Language Header 🡪 en, nl, fr etc.,

Versioning REST API

Used whenever you want to implement a breaking change in the REST calls.

Options:

* URL Versioning 🡪 Twitter
* Request Parameter Versioning 🡪 Amazon
* Request Header Versioning 🡪 Microsoft
* Media Type 🡪 GitHub

Factors to consider:

* URI Pollution 🡪 happens in URL, Req Params
* Caching 🡪 All can have complexities.
* Misuse of HTTP Headers 🡪 all can be misused.
* Can we execute the request on the browser? 🡪 Only URL, Req Param
* API Documentation 🡪 URL, Req param
* Think about versioning even before you need it.
* Have a consistent approach 🡪 One versioning approach across the projects in the enterprise.

HATEOAS = Hypermedia as the engine of Application state.

* Enhancing the REST API to tell consumers how to perform subsequent actions.
* Implementation Options:
  + Custom Format and Implementation 🡪 Difficult to maintain.
  + Use Standard Implementation 🡪 HAL (JSON Hypertext Application Language) Spring HATEOAS

Serialization

* Converting Object to Stream ex: JSON
* Jackson is the famous Java Serialization.
* We can customize the REST API Responses

Static Filtering

@JSONProperty 🡪 Customize field names

@JSONIgnore

@JSONIgnoreProperties

Dynamic Filtering

@JSONFilter with Filter Provider

HAL Explorer (Hypertext Application Language)

* Help in hyperlinking between the resources.
* Helps non-technical people too to play around.

**MICROSERVICES**

It’s an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

**Microservices in a nutshell are,**

1. **Exposed by REST**
2. **& small well-chosen deployable units**
3. **& cloud enabled**

A diagram of a service

Description automatically generated

**Problems with building Microservices:**

1. **Bounded Context – How do you identify what you should do and what you should not do?**
2. **Configuration Management.**
3. **Dynamic Scale up and Scale down.**
4. **Visibility – Centralized Logging, Monitoring etc.,**
5. **Fault Tolerance**

Solutions to the above challenges,

1. Spring Cloud Config Server 🡪 Store the configuration for different environments of all the microservices in a GIT repository and expose.

A diagram of a service

Description automatically generated

1. A diagram of a currency exchange

   Description automatically generated

Naming Server 🡪 Eureka.

Service Discovery.

Client-Side Load Balancing 🡪 Ribbon.

Easier REST Clients 🡪 Feign.

1. Visibility and Monitoring 🡪 Zipkin Distributed Tracing and Netflix API Gateway.
2. Fault Tolerance 🡪 Hystrix.

Advantages:

1. New Technology and Process adaptation
2. Dynamic Scaling
3. Faster Release Cycles

Currency Exchange Service:

http://localhost:8000/currency-exchange/from/USD/to/INR

{

"id":10001,

"from":"USD",

"to":"INR",

"conversionMultiple":65.00,

"environment":"8000 instance-id"

}

Currency Conversion Service:

http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10

{

"id": 10001,

"from": "USD",

"to": "INR",

"conversionMultiple": 65.00,

"quantity": 10,

"totalCalculatedAmount": 650.00,

"environment": "8000 instance-id"

}

Eureka:

http://localhost:8761/

Custom Routes

- http://localhost:8765/currency-exchange/from/USD/to/INR

- http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10

- http://localhost:8765/currency-conversion/v2/from/USD/to/INR/quantity/10

- http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10

http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/USD/to/INR

http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/USD/to/INR/quantity/10

http://localhost:8765/CURRENCY-CONVERSION/v2/currency-conversion/from/USD/to/INR/quantity/10

http://localhost:8765/currency-exchange/currency-exchange/from/USD/to/INR

http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10

http://localhost:8765/currency-conversion/v2/currency-conversion/from/USD/to/INR/quantity/10

**One way of deploying various language microservices (Java, Python, Go etc.,) 🡪 Containers (Docker)**

1. Create Docker images for each microservice.
2. The Docker image contains everything that a microservice needs to run.
3. Then run these docker images the same way on any infrastructure (cloud center, local machine, data center etc.,)

**Docker Hub 🡪** Docker Registry

**Image is like a class and Container is like an Object.**

Docker Architecture:

Docker Client

Docker Daemon

Containers, Local images, Image Registry

Deployments using Docker:

* Previously the applications were run using Virtual Machines for deployment. Now Docker is used.

Container 1 Container 2 Container 3

Docker Engine

Host Operating System

Cloud Infrastructure

**Docker Container pool is similar to Thread service pool (Executor Service).**

Stop => SIGTERM => gracefully shutdown.

Kill => SIGKILL => shutdown immediately.

**Distributed Tracing:-**

1. How to trace the requests across multiple microservices and identify the problem or issue?
2. All microservices send all the data logs or any to the Distributed Tracing Server into a DB.
3. Zipkin is one of the tracing server.

**Monitoring is reactive. Observability is proactive.**

1. Gather data – metrics, logs, traces 🡪 Monitoring
2. Get Intelligence – AI/Ops, Anomaly Detection 🡪 Observability
3. Open Telemetry – standard for the metrics, logs and traces.

> Spring Boot 3.x 🡪 Micrometer 🡪 logs, metrics, tracer 🡪 vendor neutral application observability façade.

Docker compose 🡪 multi-container docker applications.