**Microservices using JAVA and Spring Boot.**

Microservices -

To enable service based architecture, **monolith** applications moved towards **SOA**.

To incorporate loose coupling, **SOA** gave way for more segregated modules which are called **Microservices**.

An insight into Microservice can be obtained from -

<http://sdtimes.com/digging-into-microservices/7/>

Trying to imbibe microservice architecture in a small application -

1. 3 loosely coupled projects.

2. Using Spring Boot that provides embedded tomcat server.

Use these dependencies in the pom.xml’s -

*<dependencies>*

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-web</artifactId>*

*<version>1.2.0.RELEASE</version>*

*</dependency>*

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-jersey</artifactId>*

*<version>1.2.0.RELEASE</version>*

*</dependency>*

*</dependencies>*

3. Architecture representing 2 producers and 1 consumer.

( Idea realized with the help of - <https://dzone.com/articles/spring-boot-creating> )

Use Case -

Implement a school’s details fetching model, that comprises of 3 services -

*A. fetching details of the teacher based on subject id.*

*B. fetching details of the subject based on subject id.*

*C. fetching both the above details, based on the subject id.*

The services 1 and 2 ought to be independently up and running, and should be available for service 3 as well, when service 3 is invoked.

Implementation -

1. Producers - **teacherDetails** and **subjectDetails** are the independent projects. Their configurations are similar to each other’s. Please refer to the project structure.

2. Application.java and ApplicationConfig are the classes for Spring Boot configuration.

JerseyConfig takes care at the package level; here in our case it is enclosing POJO and REST class. The output is expected to be in simple json format.

3. Consumer - **schoolDetails** is a project that consumes the above 2 services, and during code, add the dependency either using pom.xml or build path.

A slight variation exists in the ApplicationConfig.java here i.e. the addition of restTemplate(), and as per the demo, it appears to be important for encapsulating the incoming results from various services.

Test Run -

<http://localhost:8097/teacher/getTeacher?id=100>

*{"teacherId":1001,"teacherName":"ABC","subjectId":100}*

<http://localhost:8098/subject/getSubject?id=100>

*{"subjectId":100,"subjectName":"Maths"}*

<http://localhost:8099/details/detailsApi?id=100>

*{"subjectId":100,"teacher":{"teacherId":1001,"teacherName":"ABC","subjectId":100},"subject":{"subjectId":100,"subjectName":"Maths"}}*