Micro Service, Spring Boot, Spring Cloud, **Aerovition Digital Inc.**

. The product micro-service must provide the following API calls;
- List of products
- Product details

That usse a persistence layer.

Micro-service Concerns
Handles handle the following concerns.

- Versioning
- Self-Discovery
- Security

Deployment -Calls

- Deploy it
- Upgrade it
- Call it externally (from a channel) vs. internally (from another micro-service)

To to deploy it on premise and in the cloud.

ANSWER:

START OF CODE

```
@EnableAutoConfiguration
@EnableDiscoveryClient
@Import(ProductWebApplication.class)
public class productsServer {

    @Autowired
    ProductRepository productRepository;

    public static void main(String[] args) {
        //configure / use WHY another markup  wh product-server.yml
        System.setProperty("spring.config.name", "product-server");
        SpringApplication.run(ProductServer.class, args);
    }
}
/*How to:
*Encapsulating micro-service
*SECURITY /ACCESS OF SERVICE
*/
@Service
```

```
public class WebProductService {

    @Autowired
    @LoadBalanced
    protected RestTemplate restTemplate;

    protected String serviceUrl;

    public WebProductService(String serviceUrl) {
        this.serviceUrl = serviceUrl.startsWith("http") ?
               serviceUrl : "http://" + serviceUrl;
    }

    public Product getByID(String productId) {
        Product product = restTemplate.getForObject(serviceUrl
                + "/products/{id}", Product.class, productId);

        if (product == null)
            throw new ProductNotFoundException(productId);
        else
            return product;
    }

/*
*TO RETRIEVE ALLL PRODUCTS
* INFORMATION
*Local  function.
*
*/

public Product ArrayList[] getProducts(ArrayList []Product)
     {
        ArraryList Product[] prod = new Product[]();
        Int prod_arr_length = prod.length()-1;
        for(int i=0;i<=prod_arr_lenght;i++)
        {
        String prod_name[i] = prod.getproductName(prod.productName[i]);
        String prod_id[i] = prod.getproductId(prod.productId[i]);
        String prod_details[i] = prod.getproductDetails(prod.productDetails[i]);
        System.out.println("Products Are"+  prod_id[i] + prod_name[i] +
        prod_details[i]);

        }
       return prod;
      }
}


/*
*
*
Accessing Micro-service via web.
*
*
*/
```

```
@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan(useDefaultFilters=false)  //component scanner set to false/disable
public class WebServer {

    public static void main(String[] args) {
        //configure using web-server.yml
        System.setProperty("spring.config.name", "web-server");
        SpringApplication.run(WebServer.class, args);
    }

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    public WebProductController productController() {
         // use: http://product-service
         return new WebProductController
                       ("http://PRODUCT-SERVICE");  // serviceUrl for Access.
    }
}

/*
*
*
ServiceRegistration Server.
**How to : Registry / Discovery
*
*/
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistrationServer {

  public static void main(String[] args) {
    // [spring Boot  uses to find] registration-server.yml
    System.setProperty("spring.config.name", "registration-server");
    SpringApplication.run(ServiceRegistrationServer.class, args);
  }
}


/*
*
*PRODUCT CLASS DEFINTION
*
*Micro service.
*
*
*/

public class Product{
        public String[] productName;    //{"laptop","cell","pen"}
```

```
        public String[] productId;      // {"proid1","prod2","proid3"}
        public String[] productDetails; // {"pdet1","pdet2","pdet3"}

        //constructor and setter/getters.

        //constructor. with arguments.
        public Product(String productName, String productId, String productDetails){
                this.productName = productName;
                thsi.productId  = productId;
                this.productDetails = productDetails;
                }

        public String getproductName(){
        return productName;
        }

        public String getproductId(){
        return getproductId;
        }

        public String getproductDetails(){
        return getproductDetails;
        }

        public String setproductName(String name){
        this.productName = name;
        }

        public String setproductId(String id){
        this.productId = id;
        }

        public String setproductDetails(String details){
        this.productDetails = details;
        }
}
```

RELEASE NOTES:

DEPLOYMENT:

In continuous (CI/CD)integration all micro -service co-related files and yml/properties, can be **versioned [VERSIONING]**using SVN / Github, & packaged together in Hudson/Jenkins(CI tools) for deployment per branch releases in each block point, and complied with maven adapters. For each upgrade with co-effected files, and release the same CI process is followed.

Web service gate way can provide **additional security in cloud infrastructure.**

Spring Cloud is built on Spring Boot and utilizes parent and starter POMs. POM files are:

```
<parent>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-parent</artifactId>
   <version>_Brixton_.RELEASE </version>  <!-- Name of release train -->
</parent>
<dependencies>
   <dependency>
      <!-- Setup Spring Boot -->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
   </dependency>

   <dependency>
      <!-- Setup Spring MVC & REST, use Embedded Tomcat -->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>

   <dependency>
      <!-- Spring Cloud starter -->
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter</artifactId>
   </dependency>

   <dependency>
      <!-- Eureka for service registration -->
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
   </dependency>
</dependencies>
```

Note: _Brixton_.RELEASE is the current release of co-ordinated releases -- of Spring

By default Spring Boot applications looks for an application. Properties or application.yml file for configuration.

The micro web service application looks for registration-server.properties or registration-server.yml. The configuration from registration-server.yml:

```
# Configure The  Discovery Server

eureka:

  instance:

    hostname: localhost

  client:  # Not a client, NOT to be registered  with self server  as client. The configuration specifies that self server is not a client   and stops the server process trying to register with itself.


    registerWithEureka: false

    fetchRegistry: false

server:

  port: 1111   # HTTP (Tomcat) port
```

Eureka runs on port 8761, we can use ,port 1111 for this micro service

```
/*
```

This, Product microservice uses Spring Data to implement a JPA ProductRepository and Spring RESTful API to provide a RESTful interface to product information as a straightforward Spring Boot application.

The Micro-service, registers itself with the discovery-server at start-up. See The Spring Boot startup class:

The annotations do the work:

@EnableAutoConfiguration - defines this as a Spring Boot application.

@EnableDiscoveryClient - this enables service registration and discovery. In this case, this process registers itself with the discovery-server service using its application name (see below).

@Import(ProductsWebApplication.class) - this Java Configuration class sets up everything else (see below for more details).

What makes this a microservice is the registration with the discovery-server via @EnableDiscoveryClient and its YML configuration completes the setup:

```
# Spring properties

spring:

  application:

     name: products-service
```

```
# Discovery Server Access

eureka:

  client:

    serviceUrl:

      defaultZone: http://localhost:1111/eureka/


# HTTP Server

server:

  port: 2222   # HTTP (Tomcat) port
```

Note that this file

Sets the application name as products-service. This service registers under this name and can also be accessed by this name - see below.

Specifies a custom port to listen on (2222). All my processes are using Tomcat, they can't all listen on port 8080.

The URL of the Eureka Service process - from the previous section.

CONSUMPTION OF MICRO SERVICE

---

http://localhost:1111/eureka/apps/ properties:

```
<applications>
   <versions__delta>1</versions__delta>
   <apps__hashcode>UP_1_</apps__hashcode>
   <application>
      <name>PRODUCT-SERVICE</name>
      <instance>
         <hostName>test.corp.company.com</hostName>
         <app>PRODUCT-SERVICE</app>
         <ipAddr>192,164.1.1</ipAddr><status>UP</status>
         <overriddenstatus>UNKNOWN</overriddenstatus>
         <port enabled="true">3344</port>
         <securePort enabled="false">443</securePort>
         ...
      </instance>
   </application>
</applications>
```

view: http://localhost:1111/eureka/apps/PRODUCT-SERVICE and find details for ProductService Configuration Options

Registration Time:

eureka:

instance:

leaseRenewalIntervalInSeconds: 7 //can be altered to limit 30 sec. default

---

VERSIONING.

Registration productId:

1. A process (microservice) registers with the discovery-service using a unique id. If another process registers with the same id, it is treated as a restart (for example some sort of failover or recovery) and the first process registration is discarded. This gives us the fault-tolerant system.

To run multiple instances of the same process (for load-balancing and resilience) the processes need to register with a unique id. Under Brixton release-train, that was automatic

2. Under the Angel release train, the instance-id, used by a client to register with a discovery server, was derived from the client's service name (the same as the Spring application name) and also the client's host name. The same processes running on the same host would therefore have the same id, so only one could ever register.
3. Default: Set the id property manually via the client's Eureka metadata map, like this:

eureka:

instance:

metadataMap:

instanceId: ${spring.application.name}:${spring.application.instance_id:${server.port}}

The syntax ${x:${y}} is Spring property shorthand for ${x} != null ? ${x} : ${y}.

Since the Brixton release there is also a dedicated property for this:

eureka:

instance:

instanceId: ${spring.application.name}:${spring.application.instance_id:${random.value}}

Accessing the Microservice: Web-Service

To consume a RESTful service, Spring provides the RestTemplate class. This allows to send HTTP requests to a RESTful server and fetch data in a number of formats - such as JSON and XML.

Note: Theis microservice provides a RESTful interface over HTTP, but any suitable protocol could be used. Messaging using AMQP or JMS is an obvious alternative.

Which formats can be used depends on the presence of marshaling classes on the classpath - for example JAXB is always detected since it is a standard part of Java. JSON is supported if Jackson jars are present in the classpath.

A microservice (discovery) client can use a RestTemplate and Spring will automatically configure it to be microservice aware.

The WebController / typical Spring MVC view-based controller returning HTML. The application uses Thymeleaf as the view-technology (for generating dynamic HTML)

WebServer is also a @EnableDiscoveryClient but in this case as well as registering itself with the discovery-server (which is not necessary since it offers no services of its own) it uses Eureka to locate the micro service.

Default component-scanner setup inherited from Spring Boot looks for @Component classes and, in this case, finds WebProductController and tries to create it. Use below to Disable the scanner setting:

@ComponentScan(useDefaultFilters=false).

The service-url I am passing to the WebProductController is the name the service used to register itself with the discovery-server - by default this is the same as the spring.application.name for the process which is product-service - see product-service.yml .

PRODUCT-SERVICE is a logical host (that will be obtained via discovery) not an actual host.

DEPLOYMENT:

Load Balanced RestTemplate

The RestTemplate bean will be intercepted and auto-configured by Spring Cloud (due to the @LoadBalanced annotation) to use a custom HttpRequestClient that uses Netflix Ribbon to do the microservice lookup. Ribbon is a load-balancer to manage multiple instances of a service available,).

Note: From the Brixton Release Train (Spring Cloud 1.1.0.RELEASE), the RestTemplate is no longer created automatically, which may cause confusion and potential conflicts

Note that this instance is qualified using @LoadBalanced. (The annotation is itself annotated with @Qualifier. For more than one RestTemplate bean, be sure sure to inject the right one,[caveats]

@Autowired

```
@LoadBalanced
protected RestTemplate restTemplate;
```

In the RibbonClientHttpRequestFactory:

```
String serviceId = originalUri.getHost();
ServiceInstance instance =
        loadBalancer.choose(serviceId);  // loadBalancer uses Ribbon
... if instance non-null (service exists) ...
URI uri = loadBalancer.reconstructURI(instance, originalUri);
```

The loadBalancer takes the logical service-name (as registered with the discovery-server) and converts it to the actual hostname of the chosen microservice.

A RestTemplate instance is thread-safe and can be used to access any number of services in different parts of this application (for example, a PaymentService wrapping the same RestTemplate instance accessing a payment data microservice).

Configuration

Below the relevant configuration from web-server.yml. It is used to:

Set the application name

Define the URL for accessing the discovery server

Set the Tomcat port to 3333

```
# Spring Properties
spring:
  application:
    name: web-service
# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/


# HTTP Server
```

```
server:
  port: 3333   # HTTP (Tomcat) port
```

[caveats] Notes

Spring Boot:

View Templating Engines

The Eureka dashboard (inside RegistrationServer) is implemented using FreeMarker templates but the other two applications use Thymeleaf. To make sure each uses the right view engine, there is extra configuration in each YML file.

This is at the end of registration-server.yml to disable Thymeleaf.

...

```
# Discovery Server Dashboard uses FreeMarker.  Don't want Thymeleaf templates
spring:
  thymeleaf:
    enabled: false     # Disable Thymeleaf spring:
```

Since both ProductService and WebService use thymeleaf, necessary to point each at their own templates.

FILE - product-server.yml:

```
# Spring properties
spring:
  application:
     name: product-service  # Service registers under this name
  freemarker:
    enabled: false      # Ignore Eureka dashboard FreeMarker templates
  thymeleaf:
    cache: false        # Allow Thymeleaf templates to be reloaded at runtime
    prefix: classpath:/product-server/templates/
                    # Template location for this application only
```

...

web-server.yml is similar but its templates are defined by

prefix: classpath:/web-server/templates/

Note the / on the end of each spring.thymeleaf.prefix classpath -

COMMAND-LINE EXECUTION

The jar is compiled to automatically run io.pivotal.microservices.services.Main when invoked from the command-line - see Main.java.
The Spring Boot option to set the start-class can be seen in the POM:

```
<properties>
  <!-- Stand-alone RESTFul application for testing only -->
  <start-class>io.pivotal.microservices.services.Main</start-class>
</properties>
```

ProductsConfiguration class

```
@SpringBootApplication
@EntityScan("io.pivotal.microservices.products")
@EnableJpaRepositories("io.pivotal.microservices. products ")
@PropertySource("classpath:db-config.properties")
public class ProductsWebApplication {
...
}
```

PERSISTENCE

In main configuration class for ProductsService which is a classic Spring Boot application using Spring Data, these annotations do most of the work:

@SpringBootApplication - defines this as a Spring Boot application. This convenient annotation combines

@EnableAutoConfiguration,

@Configuration and

@ComponentScan (which, by default, causes Spring to search the package containing this class, and its sub-packages, for components - potential Spring Beans: ProductController and ProductRepository) .
@EntityScan("io.pivotal.microservices.products") – (because of usage JPA), clearly specify where the @Entity classes are.

For JPA's persistence.xml or when creating a LocalContainerEntityManagerFactoryBean. Spring Boot will create this factory-bean because the spring-boot-starter-data-jpa dependency is on the class path. An alternative way of specifying where to find the @Entity classes is by using@EntityScan. This will find Product.

@EnableJpaRepositories("io.pivotal.microservices.products")- look for classes extending Spring Data's Repository marker interface and automatically implement them using JPA - see Spring Data JPA.

@PropertySource("classpath:db-config.properties") - properties to configure my DataSource – see db-config.properties.

CONFIGURING PROPERTIES

Spring Boot applications look for either application.properties or application.yml to configure themselves. Since all servers used in an application ,if are in the same project, they would automatically use the same configuration.

To avoid that, each specifies an alternative file by setting the spring.config.name property.

File: WebServer.java.

```
public static void main(String[] args) {
  // Tell server to look for web-server.properties or web-server.yml
  System.setProperty("spring.config.name", "web-server");
  SpringApplication.run(WebServer.class, args);
}
```

At runtime, the application will find and use web-server.yml in src/main/resources.

LOGGING

Spring Boot sets up INFO level logging for Spring by default. For logs for evidence of our microservices working, set level to WARN to reduce the amount of logging.

The logging level would need to be specified in each of the xxxx-server.yml configuration files. This is usually the best place to define them as logging properties cannot be specified in property files (logging has already been initialized before @PropertySource directives are processed).

All services will share the same logback.xml.
