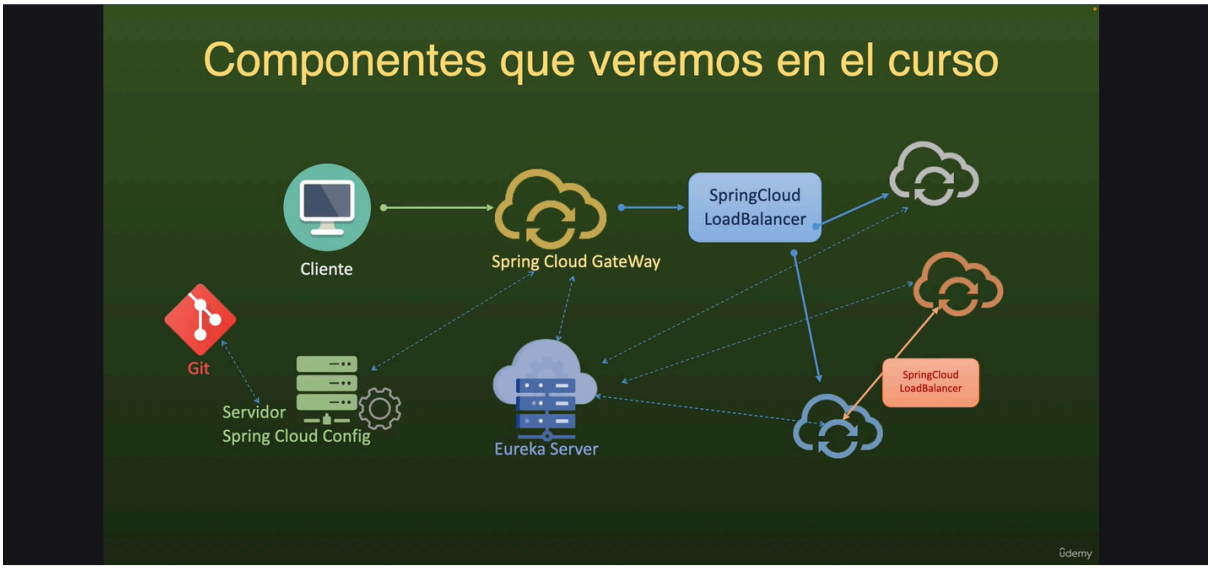
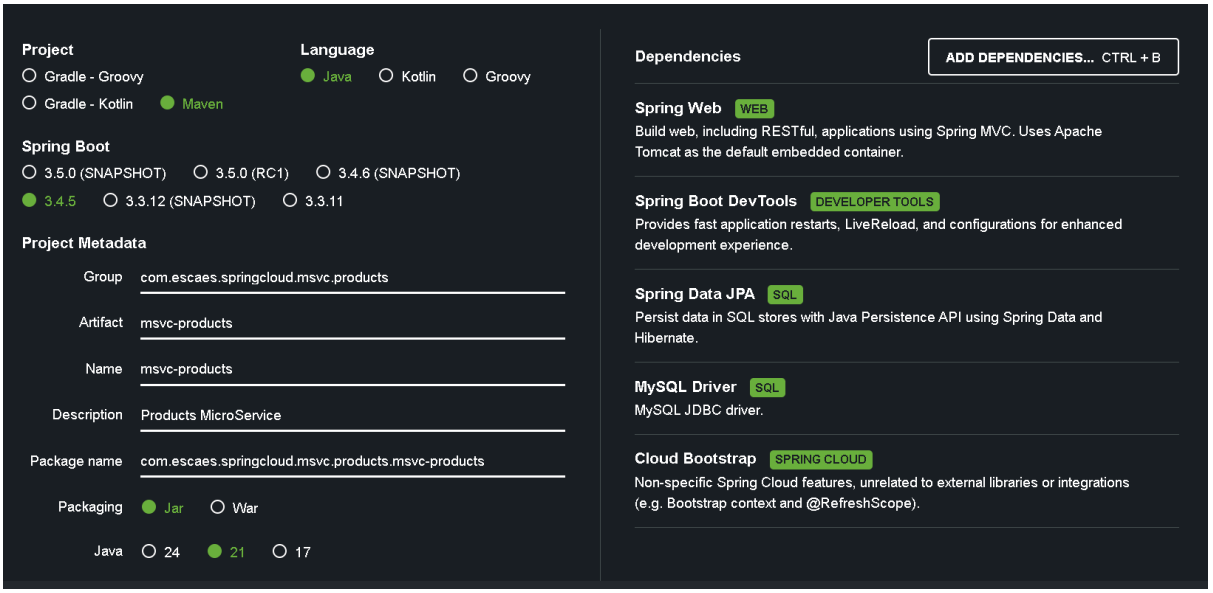
Esquema Curso de Microservicios



El cliente se conecta a nuestra **API Gateway**, la cual centraliza el acceso a todos los microservicios.
Eureka se encarga del registro y descubrimiento de los servicios, permitiendo que puedan localizarse y comunicarse entre ellos.
La comunicación entre microservicios se gestiona mediante **Spring Cloud LoadBalancer**.
Por otro lado, **Spring Cloud Config Server** administra la configuración de cada microservicio, utilizando un repositorio **HEAT** para almacenar los archivos `.properties`.

Creando MicroServicio Products



¿Por qué es importante seguir convenciones de nombres en microservicios?

1. **Organización y claridad**
    - **Al ver el nombre `com.escaes.springcloud.msvc.products`, cualquier persona sabe rápidamente:**
        - **El proyecto (`escaes`)**
        - **Que usa Spring Cloud**
        - **Que es un microservicio (`msvc`)**
        - **Y que es el de productos (`products`)**
    - **¡Evitar la confusión en proyectos grandes!**
2. **Escalabilidad de la arquitectura**
    - **Si luego creas más microservicios (usuarios, pedidos, pagos), sigues el mismo patrón:**
        - `com.escaes.springcloud.msvc.orders`
        - `com.escaes.springcloud.msvc.users`
        - `com.escaes.springcloud.msvc.payments`
    - **Así todo el ecosistema de servicios queda alineado.**
3. **Mantenimiento fácil**
    - **Si tienes que buscar un error, actualizar una dependencia, o hacer refactor, encontrar y entender el código es mucho más rápido.**
4. **Integración con otras herramientas**
    - **Muchos sistemas (Eureka, Config Server, API Gateway) basan sus rutas y configuraciones en nombres estándar de los servicios.**
    - **Si los microservicios siguen buenas convenciones, la integración es automática y menos propensa a errores.**
5. **Profesionalismo y colaboración**

- **Cuando trabajas en equipo o entregas un proyecto, seguir estándares demuestra profesionalismo.**
- **Otros desarrolladores pueden adaptarse más rápido a tu código.**

QUERY METHODS JPA

Para no realizar Querys nativas y poder usar bien las herramientas de JPA en el siguiente Link se podrán encontrar todos lo métodos con ejemplos:
JPA Query Methods :: Spring Data JPA

Inicio del Proyecto de Microservicios

Este proyecto está compuesto por dos microservicios principales:

- **MSVC-Products**
- **MSVC-Items**

El microservicio **MSVC-Products** gestiona una sencilla entidad que representa productos, siendo este el servicio principal desde donde se consumen los datos.

Por otro lado, el microservicio **MSVC-Items** no cuenta con persistencia propia, sino que se encarga de modelar y transformar los datos provenientes del microservicio de productos, haciendo uso de DTO 's para su representación.

La comunicación entre ambos microservicios se implementa de dos formas: utilizando **Feign Client** y **WebClient (WebFlux)**. Además, se ha configurado un **balanceo de carga manual**, mediante la ejecución de múltiples instancias del microservicio de productos en distintos puertos.

Cabe destacar que, en esta etapa del proyecto, **no se está utilizando un descubridor de servicios** como Eureka, ya que el balanceo se ha realizado de forma estática para fines demostrativos.

CONFIGURACIÓN Y EXPLICACIÓN MSVC-PRODUCTS

- **Dependencias (MAVEN)**

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
```

En el apartado del POM no hay mucho que especificar además que se usan estas dependencias, no se modifica nada más del POM base.

- **Entidad Products**

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    private Double price;

    @Column(name="create_at")
    private LocalDate createAt;

    @Transient// Campo transitorio: no será mapeado ni
almacenado en la base de datos.
//Se utiliza a nivel de lógica de aplicación (por ejemplo,
para exponer el puerto del servicio).
//Puede definirse como String para evitar la conversión
explícita con Integer.parseInt().

    private int port;
```

Acá además del campo Transient para el atributo port, todo sería el mismo procedimiento de creación de una Entidad usando JPA obviando que abajo están los Getters y Setters

- **Repositorio**

En este caso se utiliza JpaRepository para este microservicio, se hace la configuración habitual usando el @Entity y su extends a JpaRepository<Products,Long>

- **Servicios**

```
public interface IProductService {

    List<Product>findAll();
    Optional<Product>findById(Long id);
}
```

Y su implementación:

```
@Service //O component
public class ProductServiceImpl implements IProductService{


    private final ProductRepository productRepository;
    private final Environment environment;// Aca obtenemos el
ambiente de configuracion del proyecto donde viene
    // el properties con sus configuraciones incluyendo sus
puertos etc.

    public ProductServiceImpl(ProductRepository
productRepository,Environment environment){
        this.productRepository=productRepository;
        this.environment=environment;
    }

    @Override
    @Transactional(readOnly = true)
    public List<Product> findAll() {
        return
productRepository.findAll().stream().map(product->{
            // Si se desea evitar hacer cast, cambiar int en
la entidad por String

product.setPort(Integer.parseInt(environment.getProperty("loca
l.server.port")));
            return product;
        }).collect(Collectors.toList());
```

```
    }

    @Override
    @Transactional(readOnly = true)
    public Optional<Product> findById(Long id) {
        return productRepository.findById(id).map(product->{

product.setPort(Integer.parseInt(environment.getProperty("loca
l.server.port")));
            return product;
        });
    }
```

Nota: Es muy importante inyectar el objeto Environment de Spring Framework, ya que permite acceder a las propiedades definidas en el archivo de configuración del proyecto (**application.properties** o **application.yml**).
Esto resulta útil, por ejemplo, al utilizar stream() y map() para transformar una lista de productos, donde se puede añadir dinámicamente el número de puerto al objeto Product, accediendo a la propiedad **local.server.port** a través del Environment.

- **Controlador**

```
@RestController
public class ProductController {

    private final ProductServiceImpl serviceImpl;

    ProductController(ProductServiceImpl serviceImpl){
        this.serviceImpl=serviceImpl;
    }

    @GetMapping
    public ResponseEntity<?>list() {
        return ResponseEntity.ok(this.serviceImpl.findAll());
    }
    @GetMapping("/{id}")
    public ResponseEntity<?>details(@PathVariable Long id) {
            Product product=serviceImpl.findById(id).
```

```
            orElseThrow(()->new
ResponseStatusException(HttpStatus.NOT_FOUND,"Product Not
Found"));
        return ResponseEntity.ok(product);
    }
```

- **Properties**

No hay mucho que especificar en este pues solamente se configura la conexión a la BD usando JDBC y configurar el Hibernate en update para que persistan los datos y así no estar realizando inserts cada vez que se ejecute el microservicio.

```
spring.application.name=msvc-products
server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/db_springboo
t_cloud
spring.datasource.username=root
spring.datasource.password=escaes
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.output.ansi.enabled=ALWAYS
```

- **Configuración Launch.json(VS)**

Para realizar correctamente el balanceo de carga, es necesario ejecutar múltiples instancias del microservicio que será consumido, cada una en un puerto diferente. En este caso, partimos del puerto base `8080` y se crea una instancia adicional en el puerto `8081`.

Para lograr esto en VS Code:

1. Ve a la carpeta `.vscode` del proyecto.
2. Abre el archivo `launch.json`.
3. Duplica la configuración existente del microservicio.

4. En la segunda configuración, agrega el parámetro `"vmArgs": "-Dserver.port=8081"` para especificar el nuevo puerto.

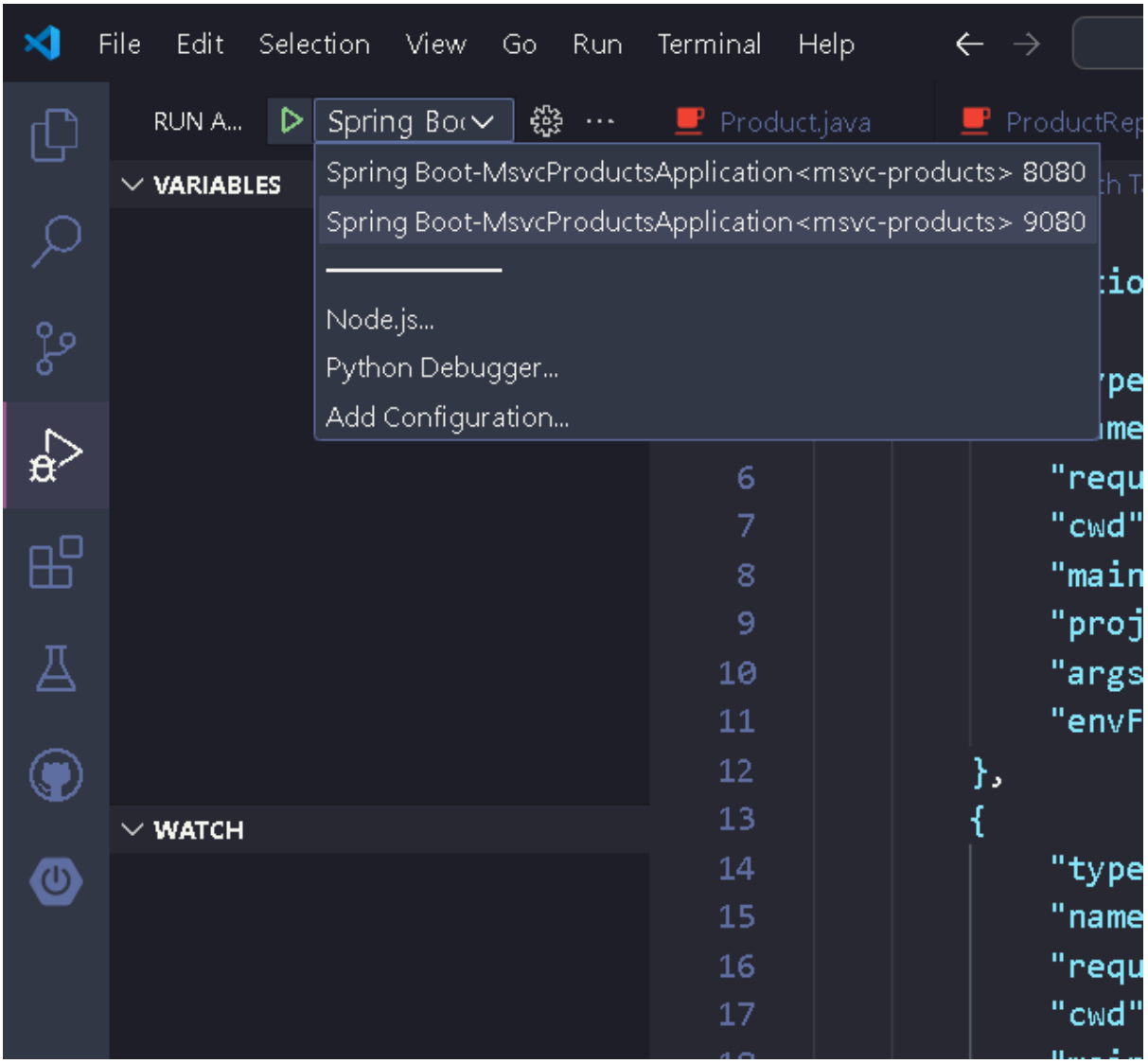
5. Opcionalmente, ajusta el valor del campo `"name"` para reflejar el puerto utilizado (por ejemplo: `"Spring Boot - msvc-products (8081)"`).

Esto permite que el mismo microservicio se ejecute en paralelo en diferentes puertos, simulando múltiples instancias para pruebas de balanceo de carga.

```
 "configurations": [
        {
            "type": "java",
            "name": "Spring
Boot-MsvcProductsApplication<msvc-products> 8080",
            "request": "launch",
            "cwd": "${workspaceFolder}",
            "mainClass":
"com.escaes.springcloud.msvc.products.msvc_products.MsvcProduc
tsApplication",
            "projectName": "msvc-products",
            "args": "",
            "envFile": "${workspaceFolder}/.env"
        },
        {
            "type": "java",
            "name": "Spring
Boot-MsvcProductsApplication<msvc-products> 9080",
            "request": "launch",
            "cwd": "${workspaceFolder}",
            "mainClass":
"com.escaes.springcloud.msvc.products.msvc_products.MsvcProduc
tsApplication",
            "projectName": "msvc-products",
            "args": "",
            "envFile": "${workspaceFolder}/.env",
            "vmArgs": "-Dserver.port=9080"
        }
    ]
```

- **Ejecución**

Para ejecutar este microservicio con sus diferentes instancias se debe ejecutar normalmente y se levantará inicialmente en el puerto 8080 o el puerto definido en el properties y después para levantar la instancia declarada en el Launch.json se debe ir al modo debug en VS y seleccionar el nombre del puerto que se desea levantar como muestra la siguiente imagen:



Por eso es importante añadir el nombre del puerto en el launch.json y continuando con el flujo después de seleccionar la instancia correspondiente se ejecuta en el mismo botón verde que se muestra a su par izquierda.

En conclusión estos serían los pasos para crear y ejecutar el microservicio productos para que eventualmente lo consuma otro como se muestra a continuación en la creación del microservicio items.

- **Resultados Esperados en Postman (MSVC-ITEMS)**