**Punto 4 — El Bucle Infinito**

Tema: Dependencias circulares, orden de inicialización y contexto de carga.

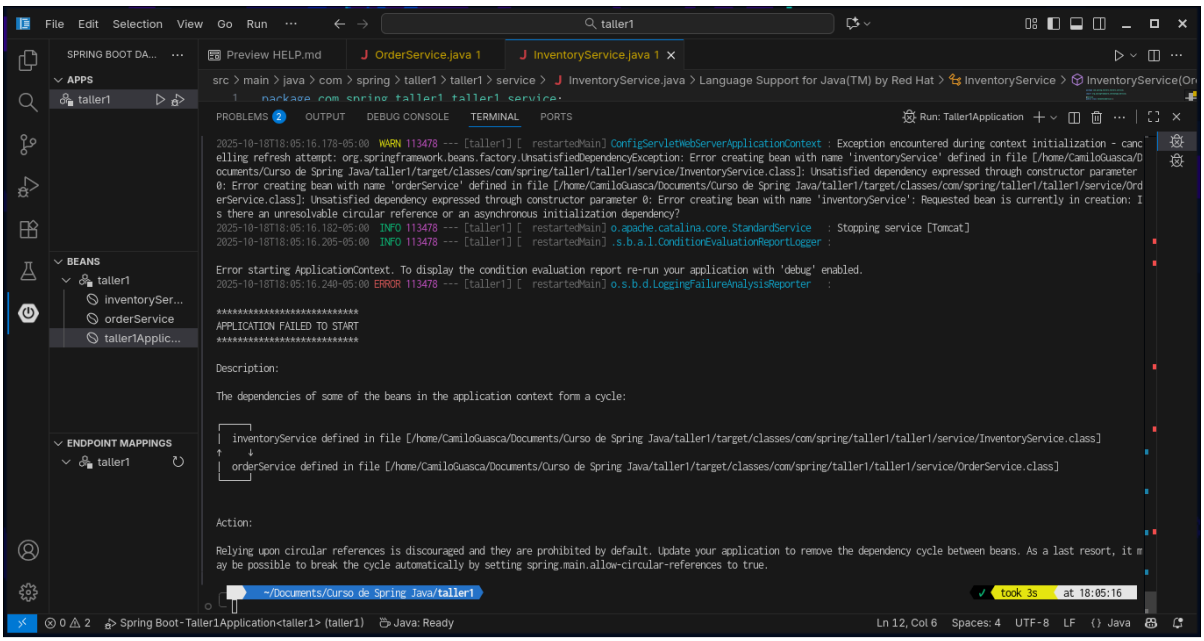
Escenario: InventoryService depende de OrderService para verificar ventas, y OrderService depende de InventoryService para validar stock. El sistema no inicia

**Retos:**

1. **Crea ambos servicios con inyección por constructor y observa el error.**

```java
package com.spring.taller1.taller1.service;

import org.springframework.stereotype.Service;

@Service
public class OrderService {
    private final InventoryService inventoryService;

    public OrderService(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
        System.out.println("OrderService inicializado");
    }

}
```

```java
package com.spring.taller1.taller1.service;

import org.springframework.stereotype.Service;

@Service
public class InventoryService {
    private final OrderService orderService;

    public InventoryService(OrderService orderService) {
        this.orderService = orderService;
        System.out.println("InventoryService inicializado");
    }
}
```

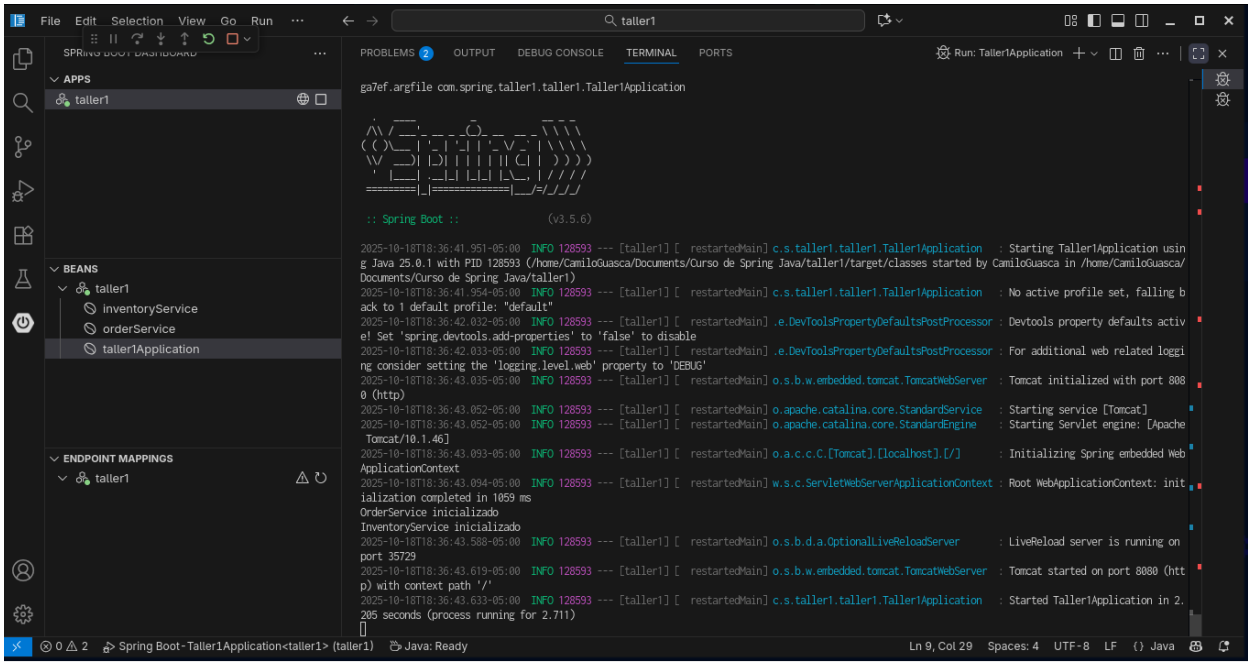



En este ejemplo podemos ver una dependencia circular mediante la inyección de dependencias por constructor. Spring intenta crear los beans detectados en este caso los @service que son servicios pero no puede porque el `InventoryService` necesita de `OrderService` y este no existe aún. Lo mismo sucede al intentar crear `OrderService` porque requiere de `InventoryService` y este no ha podido ser creado por el mismo error inicial.

2. **Cambia una de las dependencias a inyección por setter o @Lazy.**
3. **Analiza si la app arranca y qué logs aparecen.**
4. **Identifica el orden de inicialización de los beans en consola.**

```java
package com.spring.taller1.taller1.service;

import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;

@Service
public class OrderService {
    private final InventoryService inventoryService;

    public OrderService(@Lazy InventoryService inventoryService) {
        this.inventoryService = inventoryService;
        System.out.println("OrderService inicializado");
    }

}
```

```java
package com.spring.taller1.taller1.service;

import org.springframework.stereotype.Service;

@Service
public class InventoryService {
    private final OrderService orderService;

    public InventoryService(OrderService orderService) {
        this.orderService = orderService;
        System.out.println("InventoryService inicializado");
    }
}
```




Bueno ahora agregamos la anotación @Lazy en el parametro del constructor de la clase OrderService evitando asi la dependencia circular, ya que esta anotación permite que una dependendencia sea creada justo en el momento en que es solicitada para asi garantizar que exista en este caso.

5. **Propón una solución de diseño (refactorización, otro tipo de inyección, otros).**

```java
package com.spring.taller1.taller1.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class OrderService {
    private InventoryService inventoryService;

    public OrderService() {

    }

    @Autowired
    public void setInventoryService(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }
}
```

```java
package com.spring.taller1.taller1.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class InventoryService {

    private OrderService orderService;

    public InventoryService() {

    }

    @Autowired
    public void setOrderService(OrderService orderService) {
        this.orderService = orderService;
    }
}
```

Para esta parte implementamos la inyección por setter. Es decir que primero se crean los objetos vacios y después se les inyectan las dependencias para ello es usa también la anotación **`@Autowired`** que le indica a un bean al contenedor que dependencia debe inyectar automaticamente.