

epo del profesor para resolver cualquier escenario

SEC CIÓN	CONTENIDO	%NOTA
1	Arquitectura del proyecto — entender la estructura base	—
2	Los Modelos — clases Java planas	—
3	Los Repositories — datos en memoria	15%
4	Beans y Wiring — applicationContext.xml	15%
5	El Service — reglas de negocio	30%
6	Los Servlets — capa de presentación	40%
7	Orden de ejecución durante el examen	—
8	Trampas comunes y cómo evitarlas	—
9	Prompts para usar con Copilot en el examen	—

# 1. ARQUITECTURA DEL PROYECTO

*Entiende la estructura antes de tocar código*

El proyecto del profesor sigue una arquitectura en capas que se repite **sin importar el tema del examen**. Los nombres de las clases cambian (User → Device, Role → Measurement) pero los roles de cada capa son siempre los mismos. Dominar esta estructura es dominar el examen.

CAPA	PAQUETE	QUÉ HACE	COPIAR?
Model	co.icesi.model	Clases Java planas con atributos + getters/setters. Representan los datos del problema.	Adaptar
Repository	co.icesi.repositories	Almacena objetos en un ArrayList en memoria. Sin base de datos. Tiene save(), findAll(), findById().	Adaptar
Service	co.icesi.services	Contiene la lógica de negocio (validaciones, reglas). Depende de los repositories.	Adaptar
Servlet	co.icesi.servlets	Recibe peticiones HTTP (GET/POST), llama al service, genera respuesta HTML o JSON.	Adaptar
View	co.icesi.views	Genera HTML como String usando StringBuilder. El servlet la llama para construir la respuesta.	Opcional
Config	co.icesi.config	Initializer.java — arranca Spring al desplegar en Tomcat. No tocar.	■ Copiar
XML	resources/	applicationContext.xml — declara los beans y los conecta. Es el corazón del wiring.	Adaptar

■ Antes de escribir una sola línea de código, identifica en el enunciado: ¿cuáles son las dos entidades? ¿cuáles son sus atributos exactos? ¿cuáles son las reglas de negocio? Con eso en claro, completa las capas en orden de arriba hacia abajo.

## 2. LOS MODELOS

*Clases Java planas — sin anotaciones Spring*

Los modelos son clases Java simples (POJO). **No llevan ninguna anotación de Spring**. Su único trabajo es contener los datos con atributos privados y métodos get/set públicos. El profesor los define en el paquete **co.icesi.model**.

### Patrón del profesor (User.java) → Tu modelo (Device.java)

PROFESOR (User.java)	TU EXAMEN (Device.java)
private long id;	private Integer id;
private String name;	private String name;
private String username;	private String serialNumber;
private String password;	private String Ubicacion; // nombre exacto del enunciado
private List<Role> roles;	private String type;
	private String Estate; // nombre exacto del enunciado
Constructor vacío + Constructor con todos los parámetros	Constructor vacío (mínimo requerido)
Getter y setter para CADA campo	Getter y setter para CADA campo
Si tiene lista: inicializarla en el getter con new ArrayList<>()	No necesita lista (Measurement tiene el assetId)

■■ **MUY IMPORTANTE:** Usa los nombres de atributos **exactamente** como los pide el enunciado. El examen de ayer pedía **Ubicacion** (con mayúscula) y **Estate** para el estado. Si usas nombres diferentes, el wiring puede fallar o simplemente no cumplir lo pedido.

### Código completo — Device.java

```
Device.java

package co.icesi.model;

public class Device {
```

```

private Integer id;
private String name;
private String serialNumber;
private String Ubicacion; // nombre exacto pedido en el enunciado
private String type;
private String Estate; // nombre exacto pedido en el enunciado

// Constructor vacío - Spring lo necesita
public Device() {}

// Constructor completo - útil para crear datos iniciales en el repo
public Device(Integer id, String name, String serialNumber,
               String ubicacion, String type, String estate) {
    this.id = id;
    this.name = name;
    this.serialNumber = serialNumber;
    this.Ubicacion = ubicacion;
    this.type = type;
    this.Estate = estate;
}

// Getters y setters de TODOS los campos
public Integer getId() { return id; }
public void setId(Integer id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getSerialNumber() { return serialNumber; }
public void setSerialNumber(String serialNumber) { this.serialNumber = serialNumber; }
public String getUbicacion() { return Ubicacion; }
public void setUbicacion(String ubicacion) { this.Ubicacion = ubicacion; }
public String getType() { return type; }
public void setType(String type) { this.type = type; }
public String getEstate() { return Estate; }
public void setEstate(String estate) { this.Estate = estate; }
}

```

## Código completo — Measurement.java

Measurement.java

```
package co.icesi.model;
```

```
public class Measurement {  
    private Integer id;  
    private String timestamp;    // fecha y hora como String  
    private double value;  
    private String unit;        // ej: °C, bar, pH  
    private Integer assetId;    // OJO: el enunciado pide assetId, no deviceId  
  
    public Measurement() {}  
  
    public Measurement(Integer id, String timestamp, double value,  
                        String unit, Integer assetId) {  
        this.id = id;  
        this.timestamp = timestamp;  
        this.value = value;  
        this.unit = unit;  
        this.assetId = assetId;  
    }  
  
    public Integer getId() { return id; }  
    public void setId(Integer id) { this.id = id; }  
    public String getTimestamp() { return timestamp; }  
    public void setTimestamp(String timestamp) { this.timestamp = timestamp; }  
    public double getValue() { return value; }  
    public void setValue(double value) { this.value = value; }  
    public String getUnit() { return unit; }  
    public void setUnit(String unit) { this.unit = unit; }  
    public Integer getAssetId() { return assetId; }  
    public void setAssetId(Integer assetId) { this.assetId = assetId; }  
}
```

## 3. LOS REPOSITORIES

*Datos en memoria — sin base de datos · Vale 15% de la nota*

Los repositories guardan los objetos en un **ArrayList privado dentro de la clase**. No hay base de datos. Los datos viven en memoria mientras Tomcat está corriendo. Spring los gestiona como beans y por eso persisten entre peticiones HTTP.

### Análisis del patrón del profesor (UserRepository.java)

UserRepository.java — patrón del profesor

```
public class UserRepository {  
    // 1. ArrayList privado — aquí viven los datos  
    private List<User> users = new ArrayList<>();  
  
    // 2. Auto-incremental manual para los IDs  
    private long currentId;  
  
    // 3. init() — Spring lo llama al crear el bean (init-method en XML)  
    //    Aquí cargas los datos iniciales  
    public void init(){  
        User user = new User(0, 'Name', 'username', 'password', null);  
        save(user);  
    }  
  
    // 4. save() — asigna ID automático y agrega a la lista  
    public void save(User user){  
        currentId++;  
        user.setId(currentId);  
        users.add(user);  
    }  
  
    // 5. findById() — busca con stream + filter  
    public User findById(long id){  
        return users.stream().filter(u -> u.getId() == id).findFirst().get();  
    }  
  
    // 6. findAll() — devuelve toda la lista  
    public List<User> findAll() { return users; }  
}
```

## DeviceRepository.java — adaptado para el examen

### DeviceRepository.java

```
package co.icesi.repositories;

import co.icesi.model.Device;
import java.util.ArrayList;
import java.util.List;

public class DeviceRepository {

    private List<Device> devices = new ArrayList<>();
    private Integer currentId = 0;

    // El examen pide MÍNIMO 2 registros iniciales
    public void init() {
        Device d1 = new Device(0, 'Sensor Alpha', 'SN001-A', 'Planta Norte', 'sensor de temperatura', 'ACTIVE');
        Device d2 = new Device(0, 'Sensor Beta', 'SN002-B', 'Reactor 2', 'sensor de presion', 'ACTIVE');
        save(d1);
        save(d2);
    }

    public void save(Device device) {
        currentId++;
        device.setId(currentId);
        devices.add(device);
    }

    public Device findById(Integer id) {
        return devices.stream()
            .filter(d -> d.getId().equals(id))
            .findFirst()
            .orElse(null); // más seguro que .get() que lanza excepción
    }

    public List<Device> findAll() {
        return devices;
    }

    // Método adicional para eliminar (si el examen lo pide)
    public void delete(Integer id) {
        devices.removeIf(d -> d.getId().equals(id));
    }
}
```

## MeasurementRepository.java — con búsqueda por assetId

### MeasurementRepository.java

```
package co.icesi.repositories;

import co.icesi.model.Measurement;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class MeasurementRepository {

    private List<Measurement> measurements = new ArrayList<>();

    private Integer currentId = 0;

    // Datos iniciales — asociados a los devices del DeviceRepository
    public void init() {

        // assetId 1 y 2 corresponden a los devices precargados
        save(new Measurement(0, '2025-01-15T10:00:00', 23.5, 'grados C', 1));
        save(new Measurement(0, '2025-01-15T10:05:00', 45.2, 'bar', 2));
    }

    public void save(Measurement m) {

        currentId++;
        m.setId(currentId);
        measurements.add(m);
    }

    public List<Measurement> findAll() { return measurements; }

    public Measurement findById(Integer id) {

        return measurements.stream()

            .filter(m -> m.getId().equals(id)).findFirst().orElse(null);
    }

    // CLAVE: busca mediciones de un dispositivo específico
    public List<Measurement> findByAssetId(Integer assetId) {

        return measurements.stream()

            .filter(m -> m.getAssetId().equals(assetId))

            .collect(Collectors.toList());
    }
}
```

■ ■ El método `init()` debe ser público. El XML llama `init-method='init'` y si el método es privado, Spring lanza una excepción al arrancar y la aplicación no sube.

## 4. BEANS Y WIRING — applicationContext.xml

*Declara los beans y conecta las dependencias · Vale 15%*

El archivo **applicationContext.xml** es el corazón de la configuración Spring en este proyecto. Aquí le dices a Spring qué clases gestionar (beans) y cómo conectarlas entre sí (wiring). El profesor usa **setter injection** mediante la etiqueta `<property>`.

### Cómo funciona el XML del profesor

#### applicationContext.xml — patrón del profesor

```
<!-- Así define el profesor un bean con datos iniciales -->
<bean id='userRepository'
      class='co.icesi.repositories.UserRepository'
      init-method='init'>    <!-- llama init() después de crear el bean -->
</bean>

<!-- Así conecta el service con los repositories mediante setter injection -->
<bean id='userService' class='co.icesi.services.UserService'>
  <property name='repository'      ref='userRepository'>
  <property name='roleRepository' ref='roleRepository'>
</bean>

<!-- name='repository' → Spring busca setRepository() en UserService -->
<!-- ref='userRepository' → Spring inyecta el bean con ese ID -->
```

### Tu applicationContext.xml para el examen

#### applicationContext.xml — para tu examen

```
<?xml version='1.0' encoding='UTF-8'?>
<beans xmlns='http://www.springframework.org/schema/beans'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='
         http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd'>

  <!-- ① Repositorios – con init-method para cargar datos iniciales -->
  <bean id='deviceRepository'
        class='co.icesi.repositories.DeviceRepository'
```

```

        init-method='init' />

<bean id='measurementRepository'
      class='co.icesi.repositories.MeasurementRepository'
      init-method='init' />

<!-- ② Service – recibe los repos por setter injection -->
<bean id='deviceService' class='co.icesi.services.DeviceService'>
    <property name='deviceRepository'      ref='deviceRepository' />
    <property name='measurementRepository' ref='measurementRepository' />
</bean>

</beans>

```

### Regla de oro del wiring

El valor del atributo **name** en `<property>` debe coincidir **exactamente** con el nombre del setter en el Service, quitándole el prefijo 'set' y poniendo la primera letra en minúscula. Si el setter se llama **setDeviceRepository()**, el name debe ser **deviceRepository**.

En el XML	Debe existir en el Service
name="deviceRepository"	public void setDeviceRepository(...)
name="measurementRepository"	public void setMeasurementRepository(...)
name="repository"	public void setRepository(...)

### Initializer.java — copiar sin modificar

Este archivo no lo tocas. Su trabajo es registrar el contexto de Spring cuando Tomcat arranca la aplicación.

#### Initializer.java — no modificar

```

// ■ COPIAR ESTE ARCHIVO TAL CUAL DEL REPO DEL PROFESOR
public class Initializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        XmlWebApplicationContext context = new XmlWebApplicationContext();
        context.setConfigLocation('classpath:applicationContext.xml');
        servletContext.addListener(new ContextLoaderListener(context));
    }
}

```

## 5. EL SERVICE — REGLAS DE NEGOCIO

*La lógica del negocio · Vale 30% de la nota*

El Service es la clase más importante para la nota. Aquí van las validaciones y reglas que el enunciado especifica. El profesor lo conecta por **setter injection**: los repositories se inyectan mediante métodos set. Primero mira cómo lo hace el profesor, luego adapta.

### Patrón del profesor (UserService.java)

UserService.java — patrón del profesor

```
public class UserService {  
    // Dependencias declaradas como campos privados  
    private UserRepository repository;  
    private RoleRepository roleRepository;  
  
    // Setters públicos — Spring los llama para inyectar los repos  
    public void setRepository(UserRepository repository) {  
        this.repository = repository;  
    }  
    public void setRoleRepository(RoleRepository roleRepository) {  
        this.roleRepository = roleRepository;  
    }  
  
    // Métodos de negocio que usan los repos  
    public void addUser(String name, String username, String password){  
        User user = new User();  
        user.setName(name); user.setPassword(password); user.setUsername(username);  
        repository.save(user);  
    }  
    public List<User> getUsers(){ return repository.findAll(); }  
}
```

### DeviceService.java — con todas las reglas del examen

DeviceService.java — completo

```
package co.icesi.services;  
  
import co.icesi.model.Device;  
import co.icesi.model.Measurement;
```

```

import co.icesi.repositories.DeviceRepository;
import co.icesi.repositories.MeasurementRepository;
import java.util.List;

public class DeviceService {

    // ① Dependencias como campos privados
    private DeviceRepository deviceRepository;
    private MeasurementRepository measurementRepository;

    // ② Setters para que Spring inyecte las dependencias
    public void setDeviceRepository(DeviceRepository deviceRepository) {
        this.deviceRepository = deviceRepository;
    }

    public void setMeasurementRepository(MeasurementRepository measurementRepository) {
        this.measurementRepository = measurementRepository;
    }

    // ③ AGREGAR DISPOSITIVO – con todas las reglas de negocio
    public String addDevice(Device device) {
        // Regla b: nombre no vacío ni nulo
        if (device.getName() == null || device.getName().trim().isEmpty()) {
            return 'Error: El nombre no puede ser vacío o nulo';
        }

        // Regla c: serialNumber mínimo 5 caracteres
        if (device.getSerialNumber() == null ||
            device.getSerialNumber().length() < 5) {
            return 'Error: El serialNumber debe tener al menos 5 caracteres';
        }

        // Regla extra: serialNumber máximo 20 caracteres
        if (device.getSerialNumber().length() > 20) {
            return 'Error: El serialNumber no puede tener mas de 20 caracteres';
        }

        // Regla a: serialNumber único – no duplicados
        boolean serialExiste = deviceRepository.findAll().stream()
            .anyMatch(d -> d.getSerialNumber().equals(device.getSerialNumber()));
        if (serialExiste) {
            return 'Error: Ya existe un dispositivo con ese serialNumber';
        }

        // Todas las reglas pasaron – guardar
    }
}

```

```

        deviceRepository.save(device);

        return 'Dispositivo registrado exitosamente';
    }

    // ④ ACTUALIZAR ESTADO — ACTIVE / INACTIVE
    public String updateStatus(Integer id, String newStatus) {
        Device device = deviceRepository.findById(id);

        if (device == null) { return 'Error: Dispositivo no encontrado'; }

        if (!newStatus.equals('ACTIVE') && !newStatus.equals('INACTIVE')) {
            return 'Error: Estado invalido. Use ACTIVE o INACTIVE';
        }

        device.setEstate(newStatus);

        return 'Estado actualizado a ' + newStatus;
    }

    // ⑤ LISTAR TODOS
    public List<Device> getAllDevices() {
        return deviceRepository.findAll();
    }

    // ⑥ ELIMINAR — regla d: no eliminar si tiene mediciones
    public String deleteDevice(Integer id) {
        List<Measurement> mediciones =
            measurementRepository.findByAssetId(id);

        if (!mediciones.isEmpty()) {
            return 'Error: No se puede eliminar, tiene mediciones asociadas';
        }

        deviceRepository.delete(id);

        return 'Dispositivo eliminado';
    }
}

```

■ Retorna String en los métodos del service en lugar de lanzar excepciones. Es más fácil de manejar en el servlet y evita que la aplicación caiga si hay un error de validación.

## 6. LOS SERVLETS

*Capa de presentación HTTP · Vale 40% de la nota*

Los servlets son la capa más valiosa de la nota. Cada servlet atiende una URL específica y delega la lógica al service. El patrón del profesor tiene tres partes clave: **init()** obtiene el service del contexto Spring, **doGet()** devuelve HTML, **doPost()** procesa datos enviados.

### Cómo el servlet obtiene el Service — patrón del profesor

Patrón de **init()** — igual en todos los servlets

```
// Dentro del método init() de cualquier servlet:

@Override
public void init() throws ServletException {
    // Así se obtiene el contexto Spring desde el servlet
    ApplicationContext context =
        WebApplicationContextUtils.getRequiredWebApplicationContext(getServletConte
xt());
    // Así se obtiene el bean por tipo
    service = context.getBean(DeviceService.class);
}

// Este patrón es IDÉNTICO en todos los servlets - solo cambia el tipo del bean
```

### Servlet 1 — Listar dispositivos (GET con HTML)

El más parecido al del profesor. Devuelve una tabla HTML con todos los dispositivos.

ListDevicesServlet.java

```
package co.icesi.servlets;

import co.icesi.model.Device;
import co.icesi.services.DeviceService;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.context.ApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;
import java.io.IOException;
```

```

import java.util.List;

@WebServlet('/devices')          // URL: http://localhost:8080/auth/devices
public class ListDevicesServlet extends HttpServlet {

    private DeviceService service;

    @Override
    public void init() throws ServletException {
        ApplicationContext context =
            WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());

        service = context.getBean(DeviceService.class);
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        List<Device> devices = service.getAllDevices();

        StringBuilder html = new StringBuilder();
        html.append('<html><body>');

        html.append('<h2>Dispositivos IoT</h2>');

        html.append('<table border=1>');

        html.append('<tr><th>ID</th><th>Nombre</th><th>Serial</th>');
        html.append('<th>Ubicacion</th><th>Tipo</th><th>Estado</th></tr>');

        for (Device d : devices) {
            html.append('<tr>');
            html.append('<td>').append(d.getId()).append('</td>');
            html.append('<td>').append(d.getName()).append('</td>');
            html.append('<td>').append(d.getSerialNumber()).append('</td>');
            html.append('<td>').append(d.getUbicacion()).append('</td>');
            html.append('<td>').append(d.getType()).append('</td>');
            html.append('<td>').append(d.getEstate()).append('</td>');
            html.append('</tr>');
        }

        html.append('</table></body></html>');

        resp.setContentType('text/html');
        resp.getWriter().println(html.toString());
    }
}

```

## Servlet 2 — Agregar dispositivo (POST)

### AddDeviceServlet.java

```
@WebServlet('/devices/add')    // URL: http://localhost:8080/auth/devices/add

public class AddDeviceServlet extends HttpServlet {

    private DeviceService service;

    @Override

    public void init() throws ServletException {

        ApplicationContext context =

            WebApplicationContextUtils.getRequiredWebApplicationContext(getServletC
ontext());

        service = context.getBean(DeviceService.class);
    }

    // doGet - muestra el formulario HTML para agregar un dispositivo

    @Override

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)

        throws ServletException, IOException {

        resp.setContentType('text/html');

        resp.getWriter().println(

            '<html><body>' +

            '<h2>Agregar Dispositivo</h2>' +

            '<form method=POST action=/auth/devices/add>' +

            'Nombre: <input name=name/><br/>' +

            'Serial: <input name=serialNumber/><br/>' +

            'Ubicacion: <input name=ubicacion/><br/>' +

            'Tipo: <input name=type/><br/>' +

            'Estado: <input name=estate value=ACTIVE/><br/>' +

            '<button type=submit>Agregar</button>' +

            '</form></body></html>'

        );

    }

    // doPost - recibe los datos del formulario y llama al service

    @Override

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)

        throws ServletException, IOException {

        Device device = new Device();

        device.setName(req.getParameter('name'));

        device.setSerialNumber(req.getParameter('serialNumber'));
```

```

        device.setUbicacion(req.getParameter('ubicacion'));
        device.setType(req.getParameter('type'));
        device.setEstate(req.getParameter('estate'));

        String resultado = service.addDevice(device);

        resp.setContentType('text/html');
        resp.getWriter().println('<html><body><p>' + resultado + '</p>');
        resp.getWriter().println('<a href=/auth/devices>Ver lista</a></body></html>');
    };
}
}

```

## Servlet 3 — Actualizar estado (POST)

### UpdateStatusServlet.java

```

@WebServlet('/devices/status') // URL: http://localhost:8080/auth/devices/status
public class UpdateStatusServlet extends HttpServlet {

    private DeviceService service;

    @Override
    public void init() throws ServletException {
        ApplicationContext context =
            WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());
        service = context.getBean(DeviceService.class);
    }

    // doGet – formulario para actualizar el estado
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setContentType('text/html');
        resp.getWriter().println(
            '<html><body>' +
            '<h2>Actualizar Estado</h2>' +
            '<form method=POST action=/auth/devices/status>' +
            'ID del dispositivo: <input name=id type=number/><br/>' +
            'Nuevo estado: <select name=status>' +
            '<option>ACTIVE</option><option>INACTIVE</option>' +
            '</select><br/>' +
            '<button type=submit>Actualizar</button>' +

```

```

        '</form></body></html>'

    );
}

// doPost - recibe el ID y el nuevo estado, llama al service
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    Integer id = Integer.parseInt(req.getParameter('id'));
    String status = req.getParameter('status');

    String resultado = service.updateStatus(id, status);

    resp.setContentType('text/html');
    resp.getWriter().println('<html><body><p>' + resultado + '</p>');
    resp.getWriter().println('<a href=/auth/devices>Ver lista</a></body></html>');
}
}

```

# 7. ORDEN DE EJECUCIÓN EN EL EXAMEN

2 horas · Paso a paso cronometrado

■ 0–10 min  
min

## LEE EL ENUNCIADO COMPLETO

Antes de abrir VS Code, lee todo. Identifica: (1) los dos modelos con sus atributos EXACTOS incluyendo nombres especiales, (2) las reglas de negocio del service, (3) los 3 servlets que piden. Anótalas en papel o en un .txt.

■ 10–25 min  
min

## CLONA EL REPO Y ESTRUCTURA

Acepta el GitHub Classroom. Clona el repositorio. Copia la estructura de carpetas del proyecto del profesor. Copia Initializer.java, pom.xml, web.xml y applicationContext.xml como base. Verifica que el proyecto compila vacío.

■ 25–45 min  
min

## CREA LOS MODELOS

Crea las dos clases del modelo (Device.java y Measurement.java o las que pidan). Constructor vacío, constructor completo, getters y setters de TODOS los campos. Usa los nombres de atributos EXACTAMENTE como los pide el enunciado.

■ 45–70 min  
min

## CREA LOS REPOSITORIES

Crea DeviceRepository y MeasurementRepository. Copia el patrón del profesor: ArrayList, currentId, init() con 2 registros, save(), findById(), findAll(). Para MeasurementRepository agrega findByIdAssetId().

■ 70–90 min  
min

## CREA EL SERVICE Y EL XML

Crea DeviceService con los setters para inyección. Implementa TODAS las reglas de negocio del enunciado. Luego actualiza el applicationContext.xml declarando los tres beans y conectando el service con los repos.

■ 90–110  
min min

## CREA LOS 3 SERVLETS

Empieza por el servlet de lista (doGet con HTML) — es el más fácil y más parecido al del profesor. Luego el de agregar (doGet con formulario + doPost con lógica). Luego el de actualizar estado. Verifica que cada @WebServlet tenga una URL única.

---

■ 110–120  
min min

## DESPLIEGA Y PRUEBA

Haz mvn clean package. Despliega el WAR en Tomcat. Prueba cada URL en el navegador. Si algo falla, revisa primero el XML (wiring), luego los setters del service. Haz commits frecuentes.

---

## 8. TRAMPAS COMUNES Y CÓMO EVITARLAS

*Errores que hacen que la aplicación no arranque*

### XML: name no coincide con el setter

■ Si en el XML pones `name='deviceRepo'` pero el setter en el service se llama `setDeviceRepository()`, Spring no puede inyectar la dependencia. El nombre en el XML debe ser el nombre del setter sin 'set' y con minúscula inicial.

### init-method='init' pero el método no existe o es privado

■ Spring llama al método `init()` después de crear el bean. Si no existe o es `private`, la aplicación lanza excepción al arrancar. Siempre verifica que el método `init()` sea `public` y no tenga parámetros.

### Dos @WebServlet con la misma URL

■ Si dos servlets tienen `@WebServlet('/devices')`, Tomcat falla silenciosamente o solo registra uno. Cada servlet debe tener una URL completamente única.

### No copiar Initializer.java del profesor

■ Sin el `Initializer`, Spring nunca arranca su contexto y todos los servlets fallan al intentar obtener el bean. Este archivo es indispensable y no debes modificarlo.

### Usar .get() en stream sin verificar si existe

■ El método `findById()` del profesor usa `.findFirst().get()` que lanza `NoSuchElementException` si el ID no existe. Usa `.orElse(null)` y verifica `null` en el service para que la aplicación no caiga.

### Olvidar los imports en el Service

■ El service usa `List`, `Collectors`, y tus clases de `model` y `repositories`. Si faltan imports, no compila. Deja que el IDE los agregue automáticamente o verifica que estén todos al inicio del archivo.

### El WAR no incluye las clases compiladas

■ Si modificas código pero no haces `mvn clean package` de nuevo, Tomcat despliega la versión anterior. Siempre haz `clean package` y redespliega el WAR completo.

### Nombres de atributos incorrectos en el modelo

■ El enunciado puede pedir nombres específicos como 'Ubicacion' (con mayúscula) o 'Estate' en lugar de 'status'. Si los getters no coinciden con lo que el servlet espera, los datos aparecen vacíos en el HTML.

## 9. PROMPTS PARA COPILOT DURANTE EL EXAMEN

*Usa el rate con inteligencia — preguntas densas*

Durante el examen puedes usar Copilot. Recuerda que tienes rate limitado, así que haz preguntas densas. Aquí están los prompts más útiles para cada momento:

### Al inicio — mapear el enunciado

```
> @workspace Tengo un examen de Spring  
  
> El enunciado pide: [pega aquí el enunciado]  
  
> Basándote en el código del repositorio del profesor, dime exactamente qué archivos debo crear, qué debo copiar tal cual y qué debo modificar  
  
> Dame el orden recomendado.
```

### Para crear un modelo

```
> Basándote en este patrón del profesor [pega User.java], crea la clase [Nombre] con estos atributos: [lista los atributos del enunciado con sus tipos y nombres exactos]  
  
> Incluye constructor vacío, constructor completo y todos los getters/setters.
```

### Para crear un repository

```
> Basándote en este patrón [pega UserRepository.java], crea [NombreRepository] para la clase [Modelo]  
  
> Debe tener init() con 2 registros de ejemplo, save(), findAll(), findById()  
  
> Adicional necesito [describe métodos extra que pide el enunciado].
```

### Para el service con reglas

```
> Crea DeviceService que dependa de DeviceRepository y MeasurementRepository por setter injection (igual que este patrón [pega UserService.java])  
  
> Implementa estas reglas de negocio exactas: [copia las reglas del enunciado].
```

### Para el applicationContext.xml

```
> Basándote en este applicationContext.xml [pega el del profesor], crea uno nuevo para estos beans: DeviceRepository (con init-method=init), MeasurementRepository (con init-method=init), DeviceService (que depende de ambos repos por setter injection).
```

### Si algo no compila

```
> #file:[NombreArchivo.java] Este archivo tiene un error de compilación: [pega el error]

> Dime exactamente qué línea está mal y cómo corregirla

> El contexto es un proyecto Spring con XML.
```

## Si Tomcat no arranca

```
> Mi aplicación Spring falla al arrancar en Tomcat con este error: [pega el stack trace]

> El proyecto usa WebApplicationInitializer con XmlWebApplicationContext

> Dime cuál es la causa y cómo corregirla.
```

# RESUMEN EJECUTIVO

*Lo que debes recordar sí o sí*

- |   |                                 |   |
|---|---------------------------------|---|
| ① | <b>Copiar SIN tocar</b>         | Initializer.java · web.xml · pom.xml                      |
| ② | <b>Adaptar nombres</b>          | Los modelos: atributos EXACTOS del enunciado              |
| ③ | <b>Patrón Repository</b>        | ArrayList + currentId + init() + save() + findAll()       |
| ④ | <b>XML es el pegamento</b>      | name en <property> = setter sin 'set' + minúscula inicial |
| ⑤ | <b>Service = 30% nota</b>       | Todos los setters + todas las reglas de negocio           |
| ⑥ | <b>Servlets = 40% nota</b>      | init() con WebApplicationContextUtils + doGet/doPost      |
| ⑦ | <b>URL únicas</b>               | Cada @WebServlet necesita una URL diferente               |
| ⑧ | <b>Commits frecuentes</b>       | El profesor verifica el progreso — haz commit cada 20 min |
| ⑨ | <b>Probar antes de entregar</b> | Cada URL debe responder antes de hacer el push final      |

■ El examen prueba si entiendes cómo Spring conecta los objetos, no si memorizas sintaxis. Si entiendes que el XML conecta los beans mediante los setters, y que los servlets obtienen el contexto con WebApplicationContextUtils, tienes la base para resolver cualquier variación del enunciado.