Entregable 4:
CI/CD, Refactoring y code smells

Integrantes: Paulina Goycoechea y Lucía Eboli

Fecha de entrega: 18/11/2025

Introducción

El presente trabajo tiene como objetivo integrar en un proyecto real los conceptos de integración continua, refactoring y detección de code smells. Para ello se desarrolló una aplicación web denominada `Mi Playlist`, implementada en `Java` con el framework `Spring Boot`, que permite gestionar una lista personalizada de videos, incorporando funcionalidades de agregado, visualización embebida, marcado de favoritos, manejo de "likes" y persistencia local de datos.

Además del desarrollo funcional, la entrega exige la aplicación de buenas prácticas de ingeniería de software, particularmente en relación con la automatización del ciclo de vida del proyecto. En este sentido, se configuró un pipeline de CI/CD utilizando Jenkins, garantizando que cada modificación al código pase por procesos automáticos de construcción, ejecución de tests y despliegue, asegurando así calidad, repetibilidad y confiabilidad en el proceso de desarrollo.

Asimismo, se trabajó en la identificación y corrección de un code smell, aplicando una técnica de refactoring adecuada para mejorar la mantenibilidad del código. Este ejercicio permitió reforzar la importancia del diseño limpio, la modularidad y la mejora continua en proyectos de software.

Este documento presenta y justifica las decisiones técnicas adoptadas, describe la estructura del proyecto, detalla los pipelines utilizados, explica el proceso de refactoring realizado y expone los aprendizajes obtenidos a lo largo del desarrollo. Finalmente, se incluye una reflexión general sobre los contenidos del curso y algunas sugerencias para futuras ediciones.

Decisiones técnicas tomadas

El desarrollo de Mi Playlist requirió tomar una serie de decisiones técnicas orientadas a cumplir los requerimientos funcionales propuestos, garantizando al mismo tiempo simplicidad, mantenibilidad y capacidad de integrar herramientas de automatización. Cada elección tecnológica se realizó buscando equilibrio entre facilidad de implementación, robustez y pertinencia pedagógica para los objetivos del curso.

En primer lugar, se optó por implementar el backend utilizando `Java 17` junto con el framework `Spring Boot 3.5`. Esta decisión se fundamenta en que `Spring Boot` facilita significativamente la construcción de servicios REST, ofrece un ecosistema muy maduro y amplias capacidades de configuración automática, y permite estructurar el proyecto siguiendo principios de arquitectura limpia. Además, su integración natural con `Maven` simplifica la automatización del build dentro del pipeline de Jenkins.

En cuanto al frontend, se seleccionaron `React` junto con `Vite` como herramienta de construcción. `Vite` proporciona tiempos de compilación muy bajos y un entorno de desarrollo moderno, lo cual acelera el ciclo de trabajo. La interfaz se diseñó utilizando `TailwindCSS` y componentes de `shadcn/ui`, dos herramientas que permiten construir una UI más atractiva y consistente sin incrementar la complejidad del proyecto. Esta elección responde al requerimiento explícito de ofrecer una interfaz visualmente cuidada para el usuario final.

Respecto a la persistencia, se decidió no utilizar una base de datos relacional ni un motor externo, ya que la letra del proyecto no lo requería. En su lugar, se implementó un mecanismo de persistencia local mediante un archivo JSON externo (data/videos.json). Esta alternativa resulta suficiente para almacenar la información entre ejecuciones, mantiene la aplicación liviana y facilita su ejecución en cualquier entorno sin dependencias adicionales. Además, favorece el aprendizaje de serialización y lectura/escritura de archivos, manteniendo un diseño simple y ajustado al alcance planteado.

Para la integración continua y el despliegue se eligió Jenkins en su versión local. Jenkins es una herramienta ampliamente utilizada en entornos profesionales, altamente personalizable y adecuada para demostrar los conceptos de CI/CD requeridos. Su modelo de pipeline declarativo permite definir procesos automatizados de manera clara, separando las etapas de checkout del repositorio, construcción del backend, ejecución de tests, compilación del frontend y generación del paquete final de deployment. Esta configuración reproduce un flujo de trabajo realista en el que cada cambio en el código atraviesa automáticamente el ciclo de build, test y deploy.

Finalmente, se desarrollaron dos scripts de despliegue, uno para Windows y otro para Mac. Estos scripts se encargan de copiar los artefactos generados por el pipeline (el archivo JAR del backend y la carpeta compilada del frontend) hacia un directorio destinado a simular el entorno productivo. Aunque se trata de un despliegue simplificado, permite demostrar de manera efectiva el concepto de automatización de delivery solicitado en la consigna y asegura la reproducibilidad del proceso en diferentes sistemas operativos.

En conjunto, estas decisiones técnicas garantizan un proyecto coherente, mantenible y alineado con los objetivos académicos de la entrega. Permiten demostrar no solo el funcionamiento de la aplicación, sino también la correcta aplicación de los principios de integración continua y buenas prácticas de ingeniería de software.

Estructura del proyecto

La estructura del proyecto Mi Playlist fue diseñada siguiendo una separación clara entre frontend, backend y archivos de automatización, de forma de favorecer la organización, la mantenibilidad y la integración con Jenkins. El proyecto se compone de dos módulos principales: una aplicación backend desarrollada en Java con Spring Boot y una aplicación frontend desarrollada en React. Además, en la raíz del repositorio se incluyen el archivo `Jenkinsfile` y los scripts de despliegue requeridos.

En el caso del backend, ubicado en la carpeta `MiPlaylist/`, se siguió la estructura estándar que propone `Spring Boot`, organizada por paquetes según responsabilidades. Dentro del paquete controller se define la capa encargada de exponer los endpoints REST utilizados por el frontend, permitiendo operaciones de lectura, agregado, eliminación y actualización de videos. La lógica de negocio se encuentra encapsulada en service, donde se centralizan las reglas para generar IDs, obtener miniaturas desde YouTube, recuperar metadatos y administrar la lista de videos en memoria. La capa de persistencia, implementada en la carpeta repository, utiliza un archivo JSON ubicado en `data/videos.json` para almacenar los datos entre ejecuciones de la aplicación. Esta decisión permite un proyecto liviano, portable y sin dependencias externas,

manteniendo la persistencia requerida por la consigna. Finalmente, la clase model contiene la representación de la entidad Video, que define los atributos manipulados por el sistema.

El frontend, alojado en `mi-playlist-frontend/`, está construido con React y Vite. La estructura incluye una carpeta src/ con los componentes principales de la interfaz, el archivo de configuración de `TailwindCSS` y los archivos generados por Vite para el entorno de desarrollo y compilación. Este módulo es responsable de la interacción con el usuario: muestra la lista de videos, gestiona el formulario de agregado, renderiza los contenidos embebidos desde YouTube y permite realizar acciones como eliminar, dar like o marcar como favorito. La comunicación con el backend se realiza mediante peticiones HTTP hacia los endpoints REST descritos anteriormente.

A nivel de automatización, la raíz del proyecto contiene el archivo `Jenkinsfile`, que define el pipeline de CI/CD utilizado por Jenkins. Allí se especifican todas las etapas necesarias para obtener el código desde el repositorio, compilar el backend con Maven, ejecutar los tests automatizados, construir el frontend, empaquetar los resultados y ejecutar los scripts de despliegue. Junto al pipeline se incluyen los dos scripts solicitados, `deploy-windows.bat` y `deploy-mac.sh`, que realizan la copia final de los artefactos generados hacia un entorno de despliegue simulado. Estos archivos aseguran que el pipeline sea completamente reproducible tanto en sistemas Windows como en Mac/Linux.

En conjunto, la estructura del proyecto refleja una clara división de responsabilidades, donde cada módulo cumple una función específica y se integra dentro del proceso automatizado de build y despliegue. Esta organización facilita tanto el desarrollo iterativo como la demostración de los conceptos de integración continua y refactoring requeridos en la entrega.

Pipelines utilizados y justificación de cada etapa

Para automatizar el proceso de integración y despliegue continuo, se configuró un pipeline completo en Jenkins utilizando el archivo Jenkinsfile ubicado en la raíz del proyecto. La elección de un pipeline declarativo se basó en su mayor legibilidad, validación sintáctica integrada y facilidad para definir etapas de forma estructurada y segura.

El pipeline se compone de seis etapas principales, cada una cumpliendo un rol específico en el ciclo de vida de la aplicación. El objetivo general es que cualquier cambio en el repositorio pueda ser automáticamente descargado, compilado, probado, empaquetado y desplegado sin intervención manual, garantizando así una entrega continua y confiable.

Checkout del código fuente

La primera etapa del pipeline ejecuta una acción de checkout desde el repositorio GitHub donde reside el proyecto.

Esta etapa es esencial, ya que constituye el punto de partida de cualquier pipeline CI/CD: asegura que Jenkins trabaje siempre con la versión más reciente del código almacenado en main.

De esta manera se garantiza que las siguientes etapas operen sobre el estado actual del proyecto, eliminando la posibilidad de inconsistencias o ejecuciones sobre versiones locales.

Compilación del backend con Maven

La siguiente fase ejecuta `mvn clean package -DskipTests` dentro del módulo backend.
Esta etapa cumple dos funciones fundamentales:

1. Validar que el código Java compile correctamente.
   Si la compilación falla, el pipeline se detiene inmediatamente, evitando que código defectuoso avance en el proceso.

2. Generar el artefacto ejecutable (.jar).
   Este archivo representará la versión empaquetada del backend que luego será desplegada.

La arquitectura del proyecto separa backend y frontend, por lo que compilar el módulo Java de forma independiente permite identificar errores exclusivamente asociados a esa parte del sistema.

Ejecución de tests automatizados

A continuación, el pipeline ejecuta `mvn test`, que corre los tests JUnit incluidos en el proyecto.
Aunque la arquitectura propuesta y el alcance del proyecto son acotados, la presencia de al menos un test básico de carga del contexto (`contextLoads()`) permite verificar que la aplicación Spring Boot arranca correctamente, que los beans se pueden inyectar y que no existen inconsistencias graves antes del despliegue.

Desde el punto de vista de CI/CD, esta etapa resulta crítica, ya que:

- evita que una aplicación que no puede iniciar sea desplegada,
- introduce prácticas de calidad mínimas,
- posibilita la futura incorporación de más pruebas sin requerir cambios en el pipeline.

Instalación de dependencias del frontend

La cuarta etapa se ejecuta dentro del módulo frontend y realiza un `npm install`.
Este paso es indispensable porque el frontend depende de múltiples paquetes de Node.js (React, Vite, Tailwind, etc.) que deben estar presentes para poder compilar los archivos del cliente.

El pipeline, al ser reproducible, no asume que Jenkins tenga dependencias previas instaladas: cada ejecución reinstala el entorno necesario, garantizando consistencia independientemente del estado del sistema.

Construcción del frontend

Una vez instaladas las dependencias, se ejecuta `npm run build`, generando la carpeta final `dist/`, que contiene la versión optimizada del frontend apta para ser servida en producción.
Esta etapa forma parte esencial de la entrega continua porque:

- transfiere el código fuente en React a un conjunto mínimo de archivos estáticos,
- reduce el tamaño de la aplicación mediante optimizaciones,
- garantiza que la interfaz puede generarse correctamente en base al estado actual del repositorio.

Empaquetado del artefacto final

El pipeline luego crea una carpeta `deploy/` y copia dentro de ella:

- el artefacto backend (`backend.jar`), y
- el build compilado del frontend (`dist/`).

Esta etapa integra ambos módulos y los prepara como un paquete único, portable, que pueda ser desplegado tanto en Windows como en Mac/Linux. Su propósito es obtener una estructura final ordenada y lista para ser transferida a un servidor o entorno de ejecución, simulando un proceso real de entrega.

Ejecución del despliegue automático

Finalmente, el pipeline ejecuta el script correspondiente, en este caso, `deploy-windows.bat`, que copia los artefactos del proyecto hacia una carpeta externa (`C:\deploy-entregable4`).
Este paso constituye el despliegue propiamente dicho. Aunque se realiza sobre un entorno local simulado, reproduce exactamente el flujo utilizado en ambientes reales:

1. el pipeline produce artefactos,
2. los scripts se ejecutan automáticamente,
3. el nuevo build reemplaza al anterior sin intervención manual.

Esta separación entre pipeline y script facilita además la compatibilidad multiplataforma, ya que el mismo pipeline puede invocar el script equivalente (`deploy-mac.sh`) en un entorno Mac.

Justificación global del pipeline

El pipeline implementado responde plenamente a los principios fundamentales asociados a la integración y el despliegue continuo. En primer lugar, se logró una automatización completa del ciclo de vida del software, de modo que cada ejecución del pipeline puede construirse desde cero sin requerir intervención manual. Este funcionamiento garantiza que cualquier cambio introducido en el repositorio active inmediatamente los procesos de compilación, instalación de dependencias, ejecución de pruebas y empaquetado, lo cual contribuye a mantener un flujo de trabajo constante y confiable.

Asimismo, el pipeline asegura la reproducibilidad del proceso, ya que cada etapa reinstala el entorno necesario, evitando dependencias ocultas o diferencias entre ejecuciones. Gracias a ello, el sistema es capaz de detectar de manera temprana cualquier error de compilación, prueba o configuración, y detener la ejecución antes de avanzar hacia etapas posteriores. Esta característica constituye una garantía de calidad esencial dentro de un proceso de CI/CD.

Otro aspecto relevante es la integración coordinada entre el frontend y el backend. El pipeline genera de manera automática tanto el artefacto ejecutable de la aplicación Java como el build optimizado del frontend, reuniendo ambos componentes en un paquete uniforme y consistente. Esta integración simplifica el proceso de entrega y asegura que la versión desplegada siempre corresponda exactamente al código más reciente presente en el repositorio.

Finalmente, la estandarización del despliegue permite replicar el proceso en distintos entornos mediante el uso de scripts específicos para Windows y Mac/Linux. El pipeline invoca estos scripts como etapa final, garantizando un mecanismo de instalación claro, controlado y fácilmente adaptable. En conjunto, todas estas características permiten afirmar que el pipeline construido ofrece un flujo sólido, coherente con las buenas prácticas actuales de automatización y entrega continua, y cumple satisfactoriamente con los requerimientos establecidos en la consigna.

Code smell y Refactoring Aplicado

Durante el desarrollo del proyecto se identificó un code smell presente en la clase `VideoService`, específicamente en el método `addVideo`. El problema principal era la acumulación de múltiples responsabilidades dentro de un único método, lo cual constituye una violación directa del principio Single Responsibility Principle (SRP). Este síntoma es conocido en la literatura de refactoring como Long Method o Divergent Change, ya que cualquier modificación relacionada con el identificador del video, la generación de la miniatura o la obtención del artista implicaba alterar el mismo método, aumentando su complejidad y dificultando su mantenimiento.

El método original mezclaba tareas heterogéneas: asignación del identificador, construcción de la URL para la miniatura de YouTube, consulta a la API oEmbed para obtener el autor del video, procesamiento de posibles errores y, finalmente, la persistencia del objeto en la lista. Esta estructura hacía que el método fuese difícil de leer, probar y extender.

Para resolver este code smell se aplicó la técnica de refactoring Extract Method, cuyo objetivo es encapsular partes lógicas de un método complejo en funciones más pequeñas, cohesivas y con una responsabilidad claramente definida. De este modo, se redujo el acoplamiento interno del método y se mejoró tanto su claridad como su capacidad de reutilización.

Tras el refactoring, el método `addVideo` delega sus tareas secundarias en tres métodos privados: `generateId()`, `generateThumbnail(String url)` y `fetchArtistFromYoutube(String url)`. Cada uno de ellos encapsula una única responsabilidad: la generación del identificador, la obtención de la miniatura y la resolución del nombre del artista, respectivamente. Esto permitió transformar un método extenso y poco legible en una secuencia de pasos simples y declarativos.

Código original (fragmento):

```
public Video addVideo(Video video) {
    video.setId(UUID.randomUUID().toString());

    String youtubeId = extractYoutubeId(video.getUrl());
          video.setThumbnail("https://img.youtube.com/vi/"  +
youtubeId + "/0.jpg");

    try {
        String apiUrl = "https://www.youtube.com/oembed?url="
+ video.getUrl() + "&format=json";
                            ObjectMapper   mapper   =   new
ObjectMapper().findAndRegisterModules();
            Map<String, Object> data = mapper.readValue(new
URL(apiUrl), Map.class);

        String author = (String) data.get("author_name");
        video.setArtist(author != null ? author : "Unknown");

    } catch (Exception e) {
        video.setArtist("Unknown");
    }

    videos.add(0, video);
    repo.saveVideos(videos);
    return video;
}
```

Código refactorizado:

```
public Video addVideo(Video video) {
    video.setId(generateId());
    video.setThumbnail(generateThumbnail(video.getUrl()));
    video.setArtist(fetchArtistFromYoutube(video.getUrl()));

    videos.add(0, video);
    repo.saveVideos(videos);

    return video;
}

private String generateId() {
    return UUID.randomUUID().toString();
}

private String generateThumbnail(String url) {
    String youtubeId = extractYoutubeId(url);
        return  "https://img.youtube.com/vi/"  +  youtubeId  +
"/0.jpg";
}
```

```
private String fetchArtistFromYoutube(String url) {
    try {
        String apiUrl = "https://www.youtube.com/oembed?url="
+ url + "&format=json";
                            ObjectMapper mapper = new
ObjectMapper().findAndRegisterModules();

            Map<String, Object> data = mapper.readValue(new
URL(apiUrl), Map.class);
            return (String) data.getOrDefault("author_name",
"Unknown");

    } catch (Exception e) {
        return "Unknown";
    }
}
```

El resultado final es un método significativamente más claro y fácil de modificar, lo que reduce el riesgo de errores y mejora la mantenibilidad general del código. Además, el refactoring facilita la incorporación futura de nuevas transformaciones o validaciones sin comprometer la simplicidad del método principal. En consecuencia, el trabajo realizado permitió eliminar el code smell identificado y robustecer la estructura interna del servicio.

Aprendizajes obtenidos

El desarrollo de este proyecto permitió adquirir una comprensión integral sobre prácticas fundamentales de ingeniería de software que exceden la mera implementación funcional de una aplicación. En primer lugar, la experiencia reforzó la importancia de trabajar con un enfoque basado en ciclos automáticos de integración y despliegue continuo. Implementar un pipeline en Jenkins que ejecutara de manera sistemática la descarga del código, la construcción del backend en Java, la instalación y build del frontend en Node, la ejecución de pruebas automatizadas y el despliegue final, evidenció el valor de la automatización como mecanismo para garantizar calidad, reproducibilidad y rapidez en el ciclo de vida del software. A su vez, se entendió cómo los pipelines actúan como una capa de protección: ante cualquier cambio, el sistema detecta automáticamente errores en compilación o ejecución y evita que estos lleguen a producción.

En segundo lugar, el trabajo contribuyó a internalizar la relevancia de la detección y corrección temprana de code smells. Al identificar un método con responsabilidades múltiples dentro del servicio de videos y aplicar una técnica de refactoring adecuada, se hizo evidente que la calidad interna del código no solo mejora la legibilidad, sino que permite construir software más fácil de probar, extender y mantener. Este proceso puso en práctica principios de diseño como la responsabilidad única y el bajo acoplamiento, demostrando que pequeñas mejoras estructurales pueden generar un impacto significativo en la salud del proyecto.

Otro aprendizaje importante se relacionó con la persistencia y organización de la aplicación. La elección de utilizar archivos JSON externos como mecanismo de almacenamiento simplificó el manejo de datos y permitió experimentar con una capa de persistencia sencilla pero eficaz. Al mismo

tiempo, se comprendió cómo separar responsabilidades entre controladores, servicios y repositorios contribuye a una arquitectura más ordenada y escalable.

Finalmente, la experiencia adquirida en el armado completo del proyecto, incluyendo backend, frontend, persistencia, automatización, refactoring y documentación técnica, permitió apreciar la interdependencia entre distintas disciplinas del desarrollo de software. Cada parte del proceso aportó una perspectiva complementaria sobre cómo construir aplicaciones robustas y cómo sostenerlas técnicamente a lo largo del tiempo. Este enfoque holístico resultó fundamental para comprender el impacto real de las buenas prácticas profesionales en el desarrollo cotidiano y en la calidad final del producto.

Conclusión final

El curso resultó altamente enriquecedor tanto en su dimensión teórica como práctica. Entre los contenidos abordados, el tema de los lenguajes específicos de dominio (DSL) se destacó como uno de los más interesantes, al ofrecer una perspectiva distinta sobre cómo modelar soluciones utilizando lenguajes diseñados específicamente para un dominio particular. Este enfoque permitió comprender nuevas formas de abstracción y diseño, aportando herramientas conceptuales valiosas para la construcción de software más expresivo y orientado al problema.

No obstante, el presente entregable centrado en integración continua, refactoring y detección de code smells fue sin duda la instancia que más valor aportó desde el punto de vista práctico. La oportunidad de aplicar CI/CD con Jenkins, automatizar pipelines y trabajar con problemas reales de calidad de código permitió consolidar los conocimientos de una forma mucho más profunda y significativa. El proceso de construir, probar y desplegar la aplicación de manera automatizada evidenció la importancia de estas prácticas en entornos profesionales y su impacto directo en la eficiencia y mantenibilidad del software.

Uno de los aspectos más positivos del curso fue precisamente su modalidad de evaluación basada en entregables. Este enfoque nos permitió desarrollar proyectos, aplicando los conceptos en contextos prácticos y aprendiendo a través del proceso, más que mediante la memorización teórica para un parcial. Esta metodología favoreció un aprendizaje auténtico, incremental y orientado al trabajo profesional, permitiendo comprender cómo se integran las diferentes técnicas en un flujo de desarrollo completo.

Finalmente, queremos destacar especialmente la predisposición, claridad y acompañamiento de la profesora. Su orientación constante facilitó la comprensión de temas complejos y permitió avanzar con confianza en cada etapa del curso. Gracias a ello, la experiencia de aprendizaje fue aún más positiva, motivadora y acorde a las exigencias.

En conjunto, el curso dejó herramientas sólidas y aplicables que continuarán siendo fundamentales en nuestra formación y en futuros desafíos.