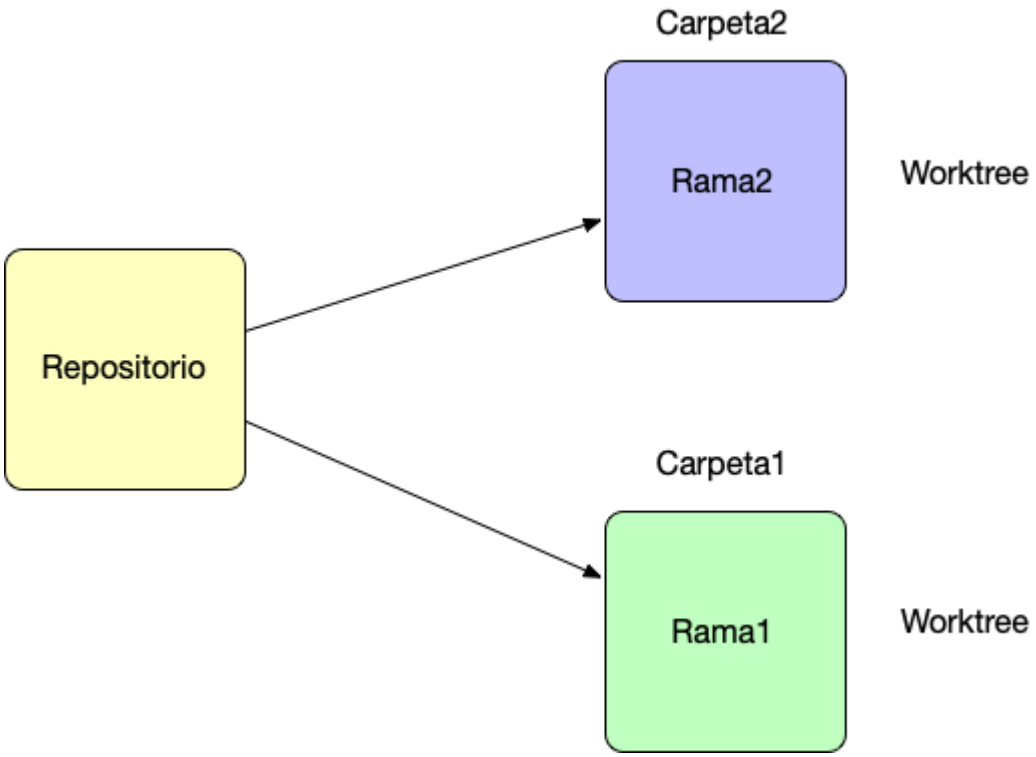
1. Introducción: qué problema resuelve git worktree

Estás desarrollando una API en Spring Boot para una pequeña biblioteca. Todo va bien hasta que aparecen dos tareas urgentes al mismo tiempo: por un lado, avanzar con el controlador de libros, y por otro, empezar el controlador de capítulos. Cambias de rama, te das cuenta de que tienes archivos a medias, haces `stash`, vuelves, pruebas algo, pierdes el hilo… y de pronto una tarea sencilla empieza a sentirse bastante caótica.

Aquí es donde `git worktree` puede ahorrarte muchos dolores de cabeza. Esta herramienta de Git permite tener varias carpetas de trabajo conectadas al mismo repositorio, pero cada una apuntando a una rama distinta. Es decir, puedes tener una carpeta para trabajar en `feature/libros` y otra para `feature/capitulos`, sin estar saltando constantemente entre ramas dentro del mismo directorio.



La ventaja principal es muy práctica: no necesitas guardar cambios incompletos con `stash` cada vez que quieres moverte a otra funcionalidad. Cada línea de trabajo vive en su propio espacio.

En este artículo vamos a verlo con un caso concreto: una API Spring Boot con dos módulos separados. Por un lado, Libros, con su `Controller`, `Service` y `Repository`. Por otro, Capítulos, también con su propia estructura.

Además, veremos cómo esta forma de trabajar puede combinarse con Claude para paralelizar agentes de desarrollo sin mezclar cambios ni perder el control del proyecto.

2. Concepto básico: cómo funciona git worktree

Concepto básico: cómo funciona git worktree

`git worktree` es una herramienta de Git que permite crear directorios de trabajo adicionales conectados al mismo repositorio. En vez de tener una sola carpeta donde vas cambiando de rama con `git checkout` o `git switch`, puedes tener varias carpetas, cada una apuntando a una rama distinta.

En el flujo tradicional, si estás en `feature/libros` y necesitas pasar a `feature/capitulos`, tienes que cambiar de rama. Si tienes cambios sin guardar, quizá debas hacer commit, usar `stash` o dejar todo en pausa. Con worktrees, la idea cambia: una carpeta puede estar en `feature/libros` y otra en `feature/capitulos`, abiertas al mismo tiempo.

Cada worktree tiene su propio estado. Puedes tener archivos modificados en uno, dependencias instaladas, configuración local e incluso un servidor de pruebas levantado, si tu proyecto lo permite. Por ejemplo, podrías ejecutar una instancia de Spring Boot para Libros y otra para Capítulos usando puertos distintos.

Importante: esto no es lo mismo que clonar el repositorio completo varias veces. Git

reutiliza la información interna cuando puede, así que el trabajo queda más ordenado y ligero.

Los comandos principales que usaremos son:

```
git worktree add
git worktree list
git worktree remove
```

Con ellos crearemos, revisaremos y eliminaremos worktrees según avance el ejemplo.

3. Preparando el proyecto Spring Boot de ejemplo

Para el ejemplo vamos a imaginar una API REST sencilla en Spring Boot para gestionar una pequeña biblioteca. Nada demasiado complejo: queremos guardar libros y, dentro de cada libro, sus capítulos.

La idea es separar el trabajo en dos funcionalidades bastante claras:

- Gestión de libros: crear, listar, buscar y eliminar libros.
- Gestión de capítulos: crear capítulos y listar los capítulos asociados a un libro.
  Una estructura típica del proyecto podría verse así:

```
src/main/java/com/ejemplo/biblioteca
├── controllers
├── services
├── repositories
└── models
```

En `controllers` estarían las clases que exponen los endpoints REST, como `LibroController` o `CapituloController`. En `services` viviría la lógica de negocio, por ejemplo `LibroService` y `CapituloService`. En `repositories` tendríamos el acceso a

datos, normalmente usando Spring Data JPA. Y en `models`, las entidades como `Libro` y `Capitulo`.

La rama principal, por ejemplo `main`, debería estar limpia antes de empezar. Desde ahí crearemos dos ramas de trabajo:

```
feature/libros
feature/capitulos
```

Esta separación ayuda a evitar conflictos innecesarios. Si una persona, o un agente, trabaja solo en libros y otra en capítulos, cada cambio tiene un espacio más claro.

Aquí encajaría muy bien una captura del árbol de carpetas inicial, o una imagen mostrando `main` como punto de partida y las dos ramas saliendo de ella. Luego cada rama tendrá su propio worktree con git worktree.

4. Creando dos worktrees: uno para Libros y otro para Capítulos

Desde la carpeta principal del proyecto, podemos crear un primer directorio de trabajo para la funcionalidad de Libros. La idea es que la rama `feature/libros` viva en su propia carpeta:

```
git worktree add ../proyecto-libros feature/libros
```

Si la rama todavía no existe, puedes crearla en el mismo paso:

```
git worktree add -b feature/libros ../proyecto-libros main
```

En `proyecto-libros` trabajaríamos todo lo relacionado con libros: `LibroController`, `LibroService` y `LibroRepository`. Esa carpeta puede abrirse en VS Code o IntelliJ IDEA como si fuera el proyecto normal, pero apuntando a su propia rama.

Luego hacemos lo mismo para Capítulos:

```
git worktree add -b feature/capitulos ../proyecto-capitulos main
```

En este segundo directorio implementaríamos `CapituloController`, `CapituloService` y `CapituloRepository`, sin mezclar esos cambios con los de libros.

La ventaja práctica aparece enseguida: puedes tener `proyecto-libros` abierto en una ventana del editor y `proyecto-capitulos` en otra. No necesitas hacer `checkout` todo el tiempo ni guardar cambios a medias.

Conviene usar nombres claros para las carpetas. Algo como `proyecto-libros` y `proyecto-capitulos` evita confusiones, especialmente si tienes varias terminales abiertas.

Para revisar cada rama, entra en su carpeta y ejecuta:

```
git status
```

Y para ver todos los worktrees activos:

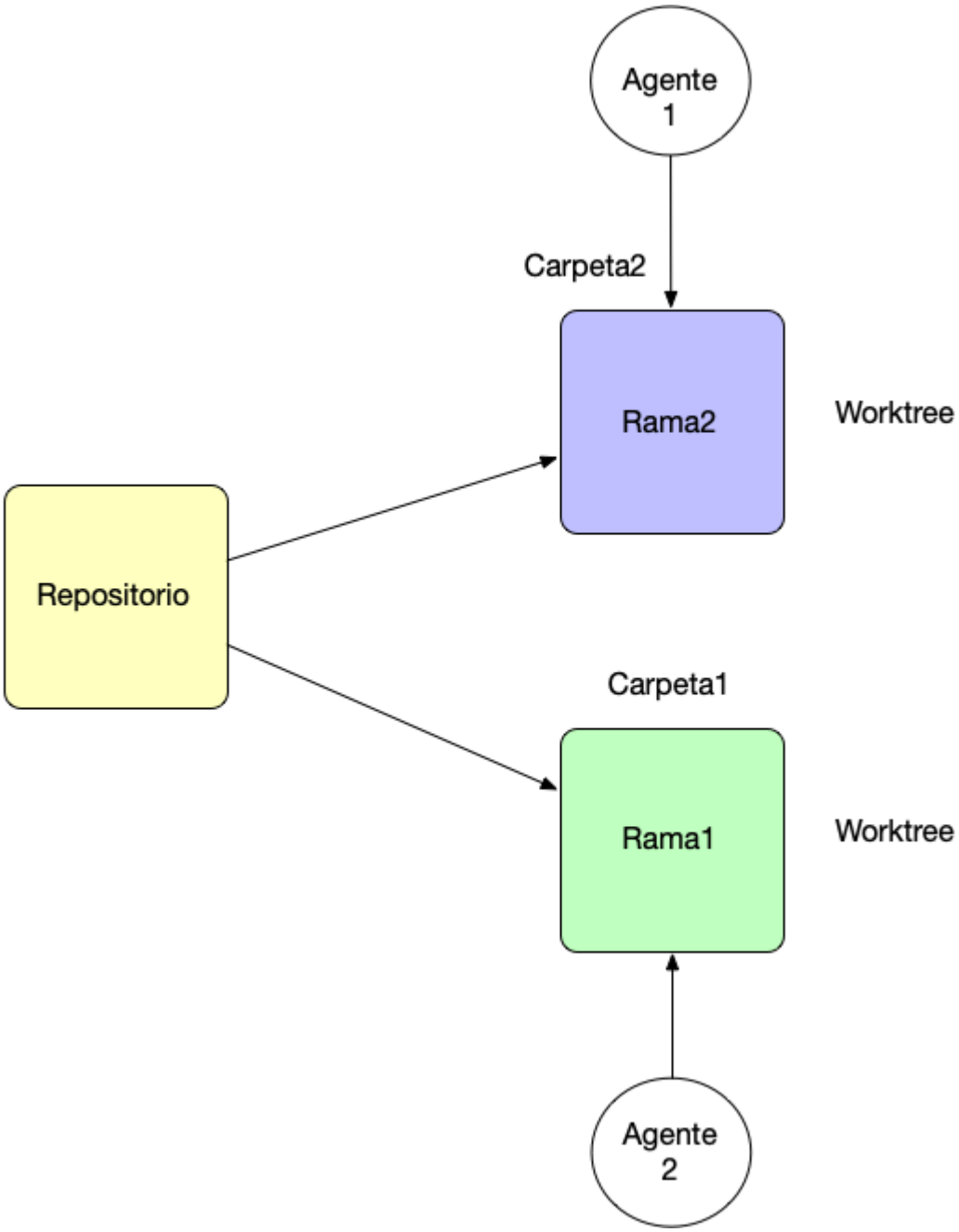
```
git worktree list
```

5. Desarrollo paralelo con Claude: agentes trabajando en worktrees separados

Aquí es donde `git worktree` se vuelve especialmente interesante si trabajas con Claude como asistente o agente de desarrollo. En lugar de pedirle que toque todo el proyecto a la vez, puedes darle una misión concreta dentro de una carpeta concreta.

Por ejemplo, puedes tener un agente trabajando en `proyecto-libros`, asociado a la rama `feature/libros`, y otro en `proyecto-capitulos`, asociado a `feature/capitulos`.

Cada uno recibe un contexto pequeño y claro, lo que reduce errores y mezclas raras entre funcionalidades.



Unas instrucciones podrían ser así:

```
En este worktree trabaja solo el módulo de Libros.
```

```
Crea endpoints para listar, buscar por id, crear y eliminar libros.
Usa LibroController, LibroService y LibroRepository.
```

Y para el otro:

```
En este worktree trabaja solo el módulo de Capítulos.
Crea endpoints para listar capítulos por libro y crear nuevos
capítulos.
Usa CapituloController, CapituloService y CapituloRepository.
```

La ventaja es que cada agente modifica archivos en su propia carpeta y en su propia rama. Hay menos interferencias, menos riesgo de pisar cambios y más claridad al revisar.

Aun así, paralelizar no significa soltar el volante. Antes de hacer commit, revisa los cambios, ejecuta los tests y valida nombres de clases, rutas, paquetes y dependencias. Si ambas ramas tocaron archivos comunes, como configuración global o modelos compartidos, tendrás que integrar con cuidado y resolver conflictos.

6. Cierre: cuándo merece la pena usar git worktree

En la práctica, git worktree merece mucho la pena cuando tienes varias líneas de trabajo abiertas: una funcionalidad nueva, un bug urgente, una prueba técnica o una refactorización que todavía no quieres mezclar con el resto. En vez de saltar de rama en rama y confiar en `stash` para guardar cambios a medias, cada tarea vive en su propia carpeta, con su propio estado y su propia rama.

En nuestro ejemplo, la API Spring Boot quedó separada en dos caminos claros: un worktree para Libros, con su `LibroController`, `LibroService` y `LibroRepository`, y otro para Capítulos, con su `CapituloController`, `CapituloService` y `CapituloRepository`. Esa separación hace que sea más fácil pensar, probar y revisar cada parte sin mezclar contextos.

Los beneficios son muy concretos: menos cambios mentales, menos uso de `stash`, más orden en el proyecto y una separación más limpia del trabajo. Y si además usas Claude como apoyo, puedes organizar agentes paralelos con más seguridad: uno trabaja en una carpeta, otro en otra, cada uno con una tarea bien delimitada.

La recomendación es simple: empieza con dos worktrees sencillos, en un proyecto pequeño, y aumenta la complejidad solo cuando el flujo te resulte cómodo.

Idea clave: git worktree no complica tu trabajo, lo separa mejor para que puedas avanzar con más calma y control.

- Curso GIT y Buenas Prácticas
- Git Branch Java y Eclipse
- Eclipse Git , Repositorios locales y remotos
- Codigo Arquitectura Java JPA Domain Driven Design
- El alzamiento de los Ingenieros de Software