Fichas Back

- Introducción al Backend y Java
- Herramientas y técnicas
- Microservicios
- Kotlin

autentia

Como en anteriores ocasiones, Autentia se embarca de nuevo en la publicación de contenidos didácticos en forma de fichas. Nuestra experiencia anterior con las fichas Agile ha sido y continúa siendo un éxito de aceptación en la comunidad.

Esto nos ha impulsado a realizar una nueva publicación con una **recopilación de buenas prácticas, patrones, principios y tecnologías para el desarrollo de software** que han ido componiendo la serie de publicaciones sobre guías técnicas que usamos en el propio On-boarding de Autentia.

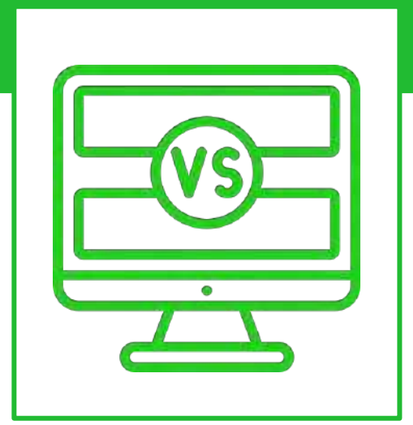
Las guías para directivos y técnicos que Autentia lleva publicando desde finales de la primavera de 2020, se han concebido ya con la incorporación de fichas que aunque integradas totalmente en cada uno de los documentos de la serie, permiten también agruparlas por temas y manejarlas de manera independiente haciendo de ellas un activo muy valioso que se puede consultar de una manera cómoda y rápida.

Estas fichas exponen conceptos, herramientas y procesos muy utilizados tanto a nivel de negocio como a nivel tecnológico pero todas con un interés y un valor apropiado que estamos seguros que la comunidad volverá a valorar.

Attribution-ShareAlike 4.0 International
(CC BY-SA 4.0)

Esta obra está licenciada bajo licencia Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Introducción al Backend y Java



¿Qué les diferencia?

Los lenguajes de alto nivel utilizan una sintaxis más parecida a la natural, a cómo nos comunicamos los humanos, mientras que los de bajo nivel utilizan unas instrucciones y sentencias más parecidas al lenguaje que hablan las máquinas.

COMPARATIVA

Ventajas de los lenguajes de alto nivel:

- El **código** es **muchísimo más mantenible** ya que si está bien escrito, es posible ,a veces a simple vista, detectar errores.
- Al ser mucho más utilizados, existen muchas más herramientas de apoyo al desarrollo y cuentan con una comunidad mucho más amplia. Esto repercute en **mayores facilidades y herramientas** de desarrollo (IDEs, sistemas de debug...)
- **Desarrollar** programas complejos **es muchísimo más rápido** ya que una sentencia de alto nivel puede englobar decenas de bajo nivel.

Ventajas de los lenguajes de bajo nivel:

- Normalmente **son más rápidos** que los lenguajes de alto nivel, y se suelen utilizar cuando el rendimiento extremo es una necesidad primordial.
- Suelen ocupar menos espacio y tienen **menos requisitos para su ejecución**, por lo que son ideales para dispositivos con poco espacio o poca memoria.

EJEMPLOS DE BAJO NIVEL

- Lenguaje de máquina.
- Lenguaje ensamblador.

EJEMPLOS DE ALTO NIVEL

- Java.
- Python.
- Swift.
- Kotlin.
- Javascript.
- C#.
- Go.
- y muchos más menos conocidos.

¿En qué consiste?

Los lenguajes de programación **compilados** son aquellos que requieren de un paso previo, en el que un compilador traduce todas las instrucciones del programa a código máquina. Un lenguaje **interpretado** realiza dicha traducción en tiempo de ejecución instrucción a instrucción.

CONSIDERACIONES

A día de hoy es raro encontrar lenguajes que se puedan considerar puramente interpretados. Lenguajes como Java, JavaScript, Python, etc., que históricamente fueron considerados como interpretados, hoy en día en realidad se podría decir que son un híbrido, ya que la mayoría realizan un paso previo en el cual traducen el código a **bytecode**, el cual es agnóstico a la plataforma en la cual se ejecuta el código y se suelen apoyar en máquinas virtuales que interpretan dicho **bytecode** y lo traducen a código máquina (por ejemplo la JVM para Java o el motor V8 para Javascript). De esta forma consiguen mayor eficiencia.

Por otra parte, en el caso de lenguajes de scripting, como bash y otros tipos de *shell* es más fácil considerarlos lenguajes interpretados.

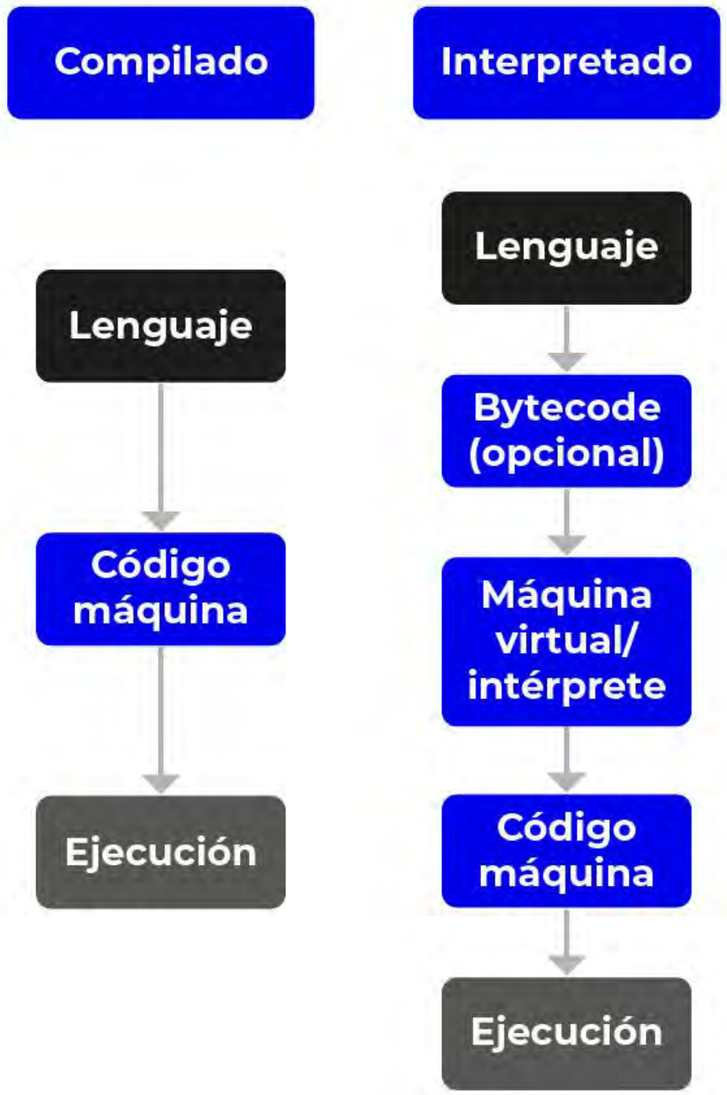
EJEMPLOS

Lenguajes compilados:

- Swift.
- C.
- C++.
- C#.
- Objective-C.
- Kotlin/Native.

Lenguajes interpretados:

- Bash, sh, zsh...
- Java.
- JavaScript.
- Python.
- Perl.
- Kotlin (JVM).



Java Virtual Machine - JVM

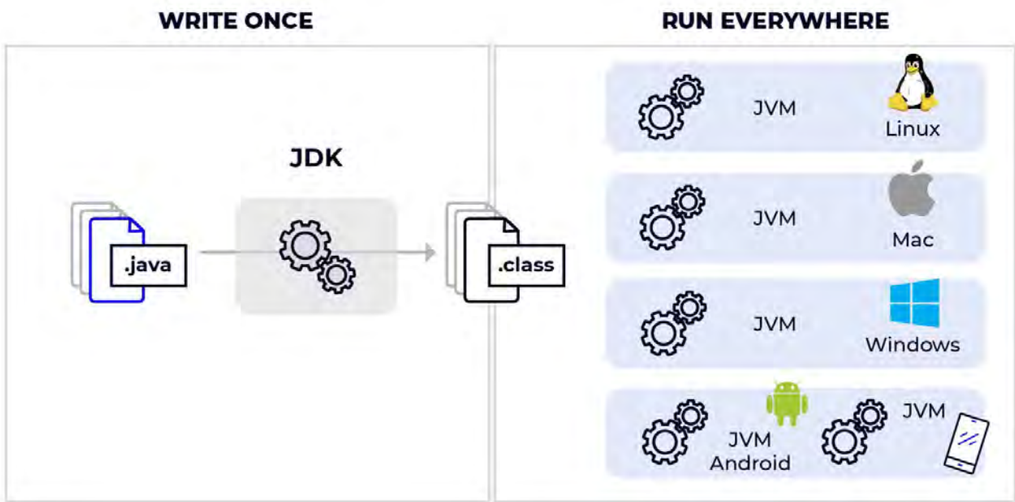
autentia

¿Qué es?

Máquina virtual que **permite ejecutar código desarrollado en Java en cualquier sistema operativo**. JVM interpreta y compila a código nativo (de la plataforma concreta de ejecución) las instrucciones expresadas en un código binario especial (**bytecode**), el cual es generado por el compilador del JDK (Java Development Kit).

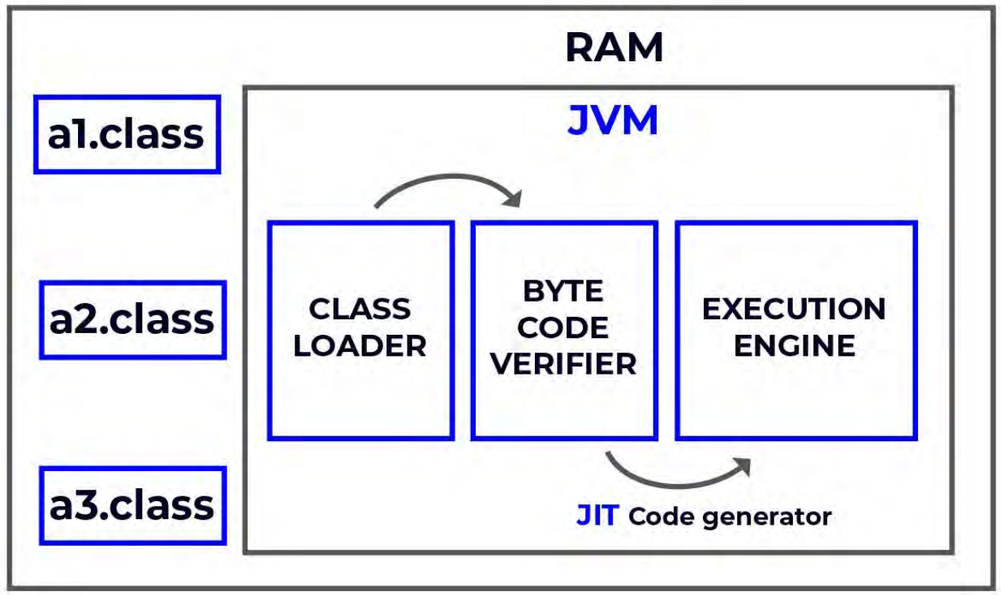
OBJETIVO

La idea cuando se desarrolló Java era **poder ejecutar el código en cualquier plataforma (Write Once Run Anywhere)**. Esto significa que un desarrollador puede escribir código Java en un sistema específico y sabe que se va a poder ejecutar en cualquier otro sin ninguna configuración adicional. La JVM es la que permite diversificar su uso y cada plataforma tiene su propia JVM, que se adapta a cada arquitectura/SO.



¿CÓMO FUNCIONA?

1. Un fichero *example.java* se compila (gracias al compilador del JDK) y se genera un fichero *example.class* que contiene el bytecode capaz de ser interpretado por la JVM.
2. Durante la ejecución del código, ***Class Loader*** se encarga de llevar los ficheros .class a la JVM que reside en la RAM del ordenador y ***ByteCode Verifier*** se encarga de verificar que el código en bytecode procede de una compilación válida.
3. El **compilador Just in Time** (esto se hace en tiempo de ejecución) compila el bytecode a código nativo de la máquina y se ejecuta directamente.



Programación Orientada a Objetos (POO)

autentia

¿Qué es?

La Programación Orientada a Objetos (POO) es un paradigma de programación, una manera de programar específica que vino a revolucionar la visión que hasta entonces se tenía en los años 80. La POO supuso un cambio a la hora de programar más cercano a un lenguaje natural que los lenguajes existentes hasta el momento. En la POO se agrupa el código en objetos individuales que poseen información y funciones.

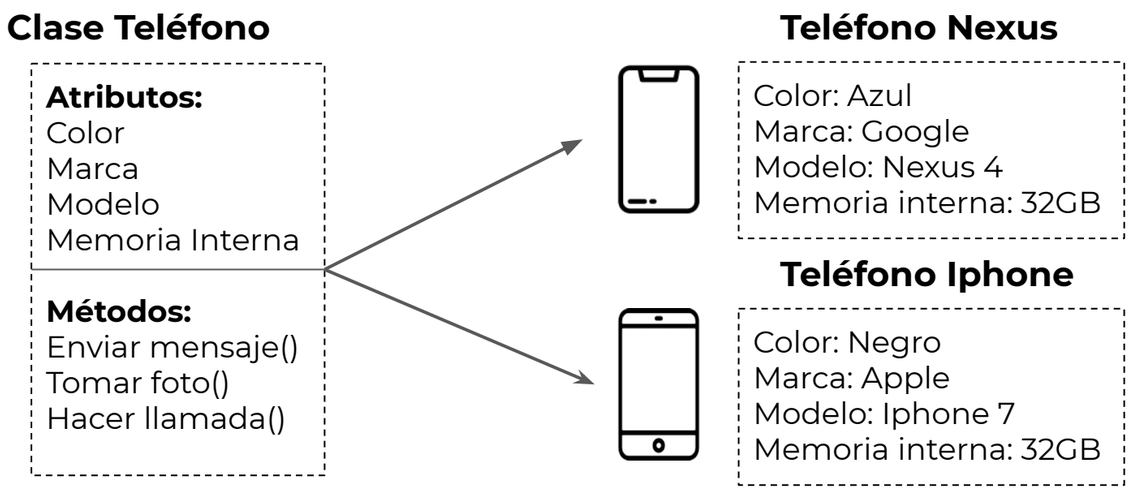
ENTENDIENDO LOS OBJETOS

Para entender qué son los **objetos** en la POO vamos a pensar en un teléfono móvil. Cualquier teléfono móvil tiene una serie de características o **propiedades** como puede ser el color, la marca, el modelo, su memoria interna, etc.

Pero además, estos tienen una serie de funcionalidades como son: enviar mensajes, hacer llamadas, tomar fotos, instalar aplicaciones. A estas funcionalidades las llamamos **métodos.**

Por último, para crear un teléfono móvil se utiliza una plantilla previamente diseñada con las características que lo defino. A esta plantilla vacía que nos va a permitir crear cualquier número de objetos "teléfono móvil", en la POO, la llamamos **clase.**

Pero en la POO, no sólo podemos definir como clases objetos físicos, sino conceptos más abstractos como por ejemplo conceptos matemáticos como una integral o una fracción. Para generar un objeto en Java:



CARACTERÍSTICAS

Las principales características más relevantes de la POO son:

- **Abstracción**: Representación de las características esenciales de algo sin incluir antecedentes o detalles irrelevantes.
- **Encapsulamiento:** Consiste en agrupar los elementos que corresponden a una misma entidad en el mismo nivel de abstracción.
- **Principio de ocultación:** Capacidad de ocultar los detalles dentro de un objeto. Protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas.
- **Herencia:** Mecanismo para compartir automáticamente métodos y atributos entre clases y subclases. De esta forma se relacionan las clases entre sí y generan jerarquías de organización.
- **Polimorfismo:** Característica que permite implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación.
- **Modularidad:** Propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **Recolección de basura:** en inglés garbage collection, es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos.

¿En qué consiste?

Paradigma de programación **declarativa** en la que las **funciones** son ciudadanas de primera clase.

CARACTERÍSTICAS

- **Funciones de primera clase y de orden superior**
  - Las funciones pueden recibir y devolver otras funciones.
  - Una función puede asignarse a una variable.
- **Funciones puras**
  - No tienen efectos secundarios.
  - Dado un parámetro de entrada devuelven siempre el mismo resultado.
- **Programación declarativa**
  - Expresamos qué queremos hacer y no el cómo.
- **No hay estructuras de control**
  - Se utiliza la recursividad para resolver problemas en los que se utilizarían estas estructuras tradicionalmente.
- **Inmutabilidad**
  - Los lenguajes puramente funcionales simulan el estado pasando datos inmutables entre las funciones.

LENGUAJES

Algunos lenguajes adoptan este paradigma por completo y todas las funciones son **puras**, lo que significa que no hay estado mutable como tal, ni se permiten efectos secundarios.

Por otra parte, algunos lenguajes, llamados **impuros** adoptan parcialmente la programación funcional, permitiendo el uso de características propias funcionales junto con las de otros paradigmas. En los impuros, podremos escribir parte del código en un estilo funcional.

Algunos ejemplos de lenguajes funcionales son:

Puros:

- Haskell.
- Miranda.

Impuros:

- Scala.
- Python.
- Java (a partir del 8).
- Kotlin.
- Rust.

¿Qué es?

La programación reactiva es un paradigma basado en el desarrollo declarativo por contra al tradicional, que es imperativo. Se trata de funcionar de una forma **no bloqueante, asíncrona y dirigida por eventos**.

¿QUÉ PROBLEMA RESUELVE?

Se trata de evitar ciertos problemas que se han visto que pueden suceder con las arquitecturas tradicionales, que suelen funcionar de una forma bloqueante. Esto puede en ocasiones desaprovechar la CPU, ya que los hilos se encuentran bloqueados por entrada salida (ir a base de datos, consultar el API de un tercero...), incluso puede llegar a la saturación del sistema y finalmente su caída con volúmenes de carga altos. La programación reactiva se sustenta en la base de **NIO**. Con NIO en vez de un pool de hilos en la que cada hilo procesa una petición de inicio a fin, vamos a tener hilos workers. Estos workers van a coger las peticiones de los usuarios y en vez de esperar cuando se bloqueen, van a utilizarse para servir otras peticiones, aprovechando al máximo nuestra CPU. Es decir, vamos a tener un escalado vertical óptimo. Tanto en on premise como en cloud esto puede **suponer un ahorro en costes de infraestructura**. Lo recomendable es que el número de workers sea igual que el número de cores que tenga nuestra CPU.

Con las nuevas arquitecturas, como por ejemplo los microservicios, nos podemos ver en la necesidad de hacer múltiples llamadas entre ellos, lo que puede desembocar en una maraña de mensajes que tienen que ir en cierto orden para al final producir una respuesta. Si a eso le sumamos el protocolo de comunicaciones HTTP (que es síncrono) podemos tener problemas de rendimiento.

En el 2014 surge el manifiesto de los sistemas reactivos. Podemos ver conceptos como que los sistemas tienen que ser **responsivos** (responder rápido en la medida de lo posible), **resilientes** (tolerante a fallos), **elásticos** (adaptable a carga) **y orientados a mensajes** (de forma asíncrona). La programación reactiva nos ayuda solo en el último de estos pasos y en algunos casos en el primero. Uno de los conceptos claves será establecer un **mecanismo de backpressure**, que nos va a permitir limitar los mensajes por parte de los productores para que los consumidores no se saturen.

Se está viendo cada vez una adopción más de este paradigma y solo hace falta mirar lenguajes como Java, que desde la versión 9 incorpora Reactive Sreams o frameworks como Spring que también lo incluyen en su módulo webflux. La programación reactiva no debe usarse para todo. Se recomienda para escenarios con tiempos de respuesta superiores al segundo cuyo tiempo se pierde casi en la totalidad en peticiones bloqueantes.

Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.

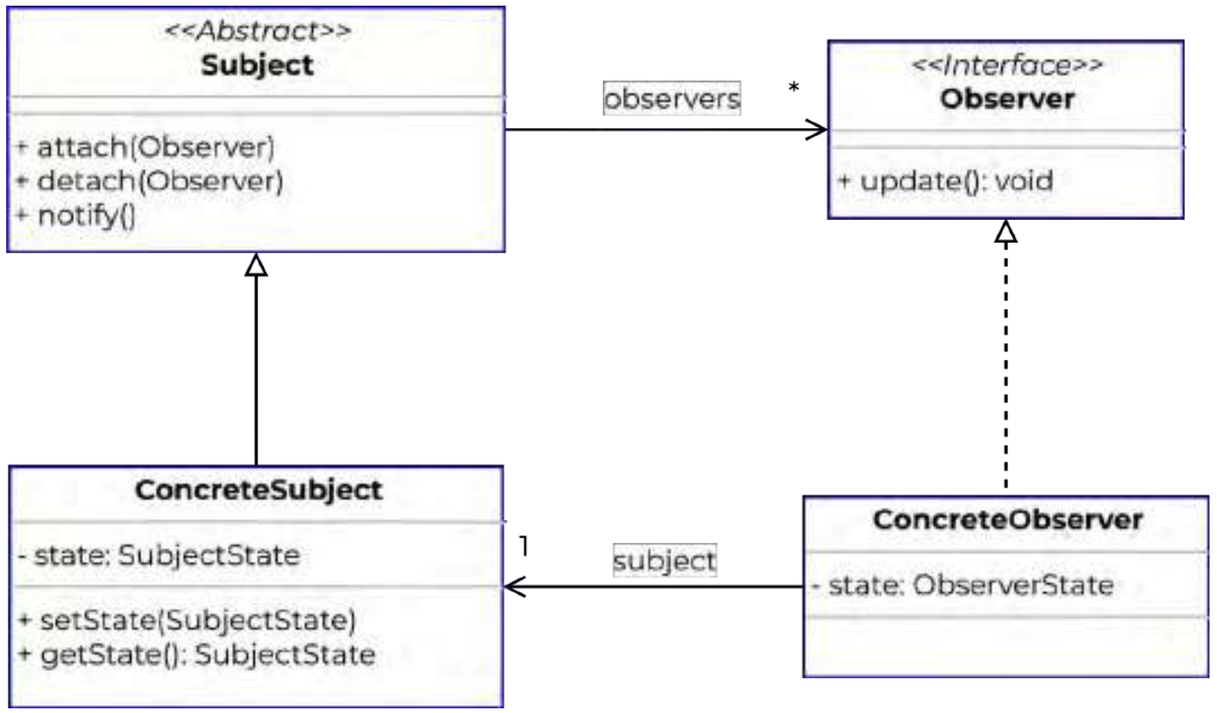
CONCEPTO

Dado el diagrama, tenemos una clase abstracta, **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores, invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una manera de implementar el patrón. Se observa que el ConcreteObserver tiene una referencia al ConcreteSubject, lo cual le permite obtener su estado. De esta manera, cuando el Subject notifique a sus Observers, esta implementación en particular, tendrá una referencia directa al Subject para poder reaccionar en base a los cambios sucedidos.

Esta no es la única forma de diseñar el patrón Observer, pero sí es la clásica descrita por el *GoF* (Gang of Four).



El sistema de publicaciones

Oracle con la publicación de Java SE 9 en Septiembre de 2017 no sólo introdujo como mejora la Modularidad en Java, también decidió apostar por las versiones de Java menores que no disponen de soporte extendido (LTS) y que desde entonces se publican cada 6 meses, siendo la última Java SE 14. La próxima versión LTS será Java SE 17 en Septiembre de 2021. A continuación, destacamos las novedades de las últimas versiones publicadas:

NOVEDADES JAVA 12

Las novedades más destacadas son:

El recolector de basura Shenandoah: con la pretensión de mejorar el rendimiento de las aplicaciones.

230: Microbenchmark Suite

Las expresiones Switch: Se trata de una mejora que nos permite eliminar varias sentencias *if else* encadenadas en la expresión Switch.

Colectores Teeing: para enviar un elemento de un Stream a dos Streams.

Formato de número compacto: Ahora se puede expresar un número en formato compacto.

Suite de Microbenchmark: añade una suite básica de Microbenchmark al código fuente del JDK que facilita a los desarrolladores ejecutar Microbenchmark existentes o crear nuevos.

Fuente: https://docs.oracle.com/en/java/javase/12/

NOVEDADES JAVA 13

Las novedades más destacadas son:

Bloques de texto: utilizando la triple comilla doble (“””) se permite identificar grandes bloques de texto que facilitan la visualización del código.

Expresiones Switch mejoradas: ahora las expresiones switch pueden también devolver un valor en lugar de sentencias.

Archivos CDS Dinámicos: Extiende el uso de la aplicación CDS para permitir el archivado dinámico de clases al final de la ejecución de la aplicación Java.

Uncommit de la memoria no utilizada: Mejora ZGC para devolver la memoria de almacenamiento dinámico no utilizada al sistema operativo.

Reimplementación de la API Socket Legacy: Permite reemplazar la implementación subyacente utilizada por las API con una implementación más simple y moderna que sea fácil de mantener y depurar.

Fuente: https://docs.oracle.com/en/java/javase/13/

NOVEDADES JAVA 14

Las novedades más destacadas son:

Records. Sin duda alguna, la novedad más importante de esta versión de Java. Los registros son clases que no contienen más datos que los públicos declarados y permite reducir considerablemente el código en algunas clases.

Excepciones NullPointerException más útiles

Pattern Matching para el operador instanceof

Bloques de texto: se definen nuevos caracteres de escape.

Recolector de basura ZGC para Windows y MacOS

Expresiones switch en modo vista

Herramienta de Packaging

Actualización de MappedByteBuffer para soportar el acceso a la memoria no volátil (NVM).

Fuente: https://docs.oracle.com/en/java/javase/14/

¿Qué es?

Es una **JVM** y **JDK** basada en HotSpot/OpenJDK e implementada en Java. Soporta lenguajes de programación adicionales y modos de ejecución, como la compilación ***ahead-of-time*** que permite un **tiempo de arranque más rápido** en aplicaciones Java, resultando en ejecutables que **ocupan menos memoria.**

OBJETIVOS

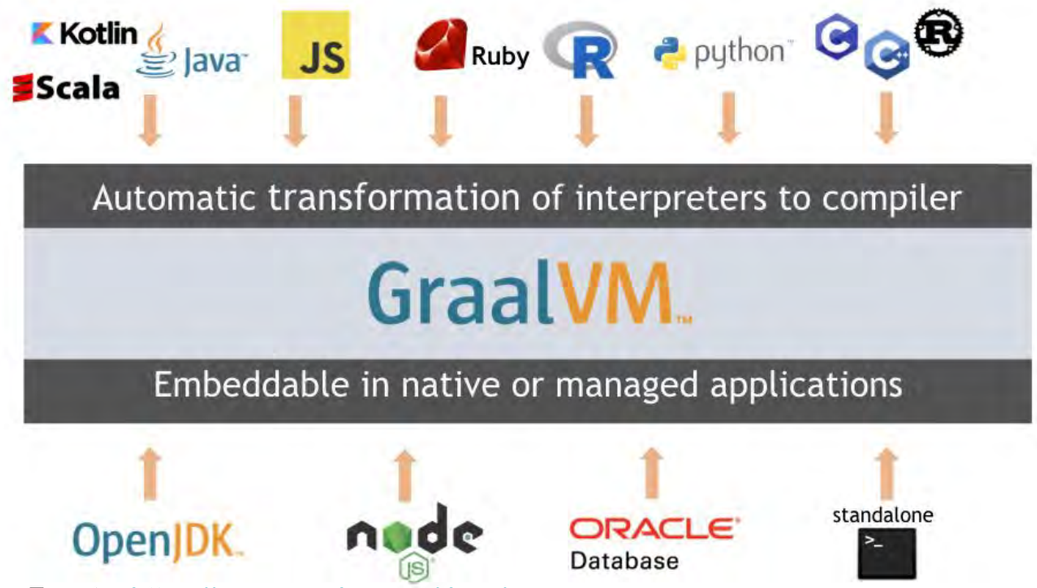
- Mejorar el **rendimiento** de los lenguajes basados en la máquina virtual de Java haciendo que tengan un rendimiento similar a los lenguajes nativos.
- **Reducir el tiempo de arranque** de las aplicaciones de la JVM mediante la compilación *ahead-of-time* (antes de tiempo) con GraalVM Native image.
- Permitir la integración de GraalVM en Oracle Database, OpenJDK, Node.js, Android/iOS y otros similares.
- Permitir utilizar código de cualquier lenguaje **en una única aplicación.**
- Incluir una colección de herramientas, fácilmente extensible, para la programación de **aplicaciones políglotas**.

COMPONENTES

- GraalVM Compiler: se trata de un compilador **JIT** para Java
- GraalVM Native Image: permite la compilación ***ahead-of-time***
- Truffle Language Implementation Framework: depende de GraalVM SDK y permite implementar **otros lenguajes** en GraalVM
- Instrumentation-based Tool Support: soporte para **instrumentación dinámica**, que es agnóstica del lenguaje.

LENGUAJES Y RUNTIMES

- GraalVM JavaScript: runtime de **JavaScript** ECMAScript 2019, con soporte para Node.js.
- TruffleRuby: implementación de **Ruby** con soporte preliminar para Ruby on Rails.
- FastR: implementación del lenguaje **R**.
- GraalVM Python: implementación de **Python 3**.
- GraalVM LLVM Runtime (SuLong): implementación de un interprete de bitcode **LLVM**.
- GraalWasm: implementación de **WebAssembly**.



Fuente: https://www.graalvm.org/docs/

¿En qué se diferencian?

Las interfaces en Java definen comportamientos, ofreciendo un catálogo de acciones posibles a los clientes que la utilizan. Una clase puede implementar esta interfaz, definiendo internamente cómo realizará las acciones ofrecidas.

¿CÓMO DEFINO UNA INTERFAZ?

Una interfaz se define así:

```
interface <nombre-interfaz> {
    // Definir firma de métodos aquí.
}
```

Luego, cualquier clase la puede implementar. Se verá obligada a implementar los métodos de la interfaz:

```
class <nombre-clase> implements <nombre-interfaz> {
    // Implementar los métodos de <nombre-interfaz>
}
```

Una clase puede implementar varias interfaces. A su vez, una interfaz puede extender de otras interfaces, agrupando varias firmas en una sola.

Los métodos de una interfaz pueden definir una implementación por defecto:

```
public default void greet() {
    System.out.println("Hola")
};
```

¿PARA QUÉ NOS SIRVE?

Las interfaces abstraen comportamientos que luego cualquier otra clase puede definir. Al momento de desarrollar, nos desliga de saber qué clase específica estamos utilizando, preocupándonos sólamente por los comportamientos que nos ofrece la interfaz.

```
interface Legs {
    void walk(NerveSignal signal);
}

class HidraulicLegs implements Legs {
   public void walk(NerveSignal signal) {
       // Activar mecanismos hidráulicos para
caminar
   }
}

class BiologicalLegs implements Legs {
   public void walk(NerveSignal signal) {
       // Activar músculos para caminar
   }
}

Human humanA = new Human(BiologicalLegs());
Human humanB = new Human(HidraulicLegs());

humanA.walk();
humanB.walk();
```

```
class Human {

   private Legs legs;

   public Human(Legs legs) {
       this.legs = legs
   }

   public void walk() {
       this.legs.walk()
   }
}
```

¿Qué son?

Las anotaciones son una característica de Java que nos permite asociar un elemento de nuestro código a una serie de metadatos. Estos metadatos son utilizados por el compilador o durante la ejecución del programa, donde otros frameworks y librerías se pueden aprovechar para generar código o realizar otras operaciones.

¿CÓMO DECLARAR UNA ANOTACIÓN?

Una anotación se declara con **@** seguido por su nombre. **Debe preceder el elemento que queremos anotar**. Por ejemplo, un caso sencillo es la anotación **@Override**, indicando que el elemento siguiente sobreescribirá un elemento de la superclase donde está definida.

Algunas anotaciones pueden tener elementos asignables. En este caso se definen entre paréntesis después del nombre de la anotación:

@Author(nombre="José", apellido="Palacios")

Otras características:

- Se pueden declarar varias anotaciones para un mismo elemento.
- Normalmente una anotación ocupará su propia línea, pero es meramente, una convención de estilo.

```
@Override
@Author(nombre="Juan", apellido="Palacios")
private String title;
```

EJEMPLO

Lombok es una librería que ofrece anotaciones para generar código. Una de sus anotaciones, @Getter, se asocia a una variable dentro de una clase:

```
public class LombokExample {

    @Getter
    private int someField;

 }
```

Al compilarla, se añadiría un método a la clase compilada:

```
public class LombokExample {

    private int someField;

    public int getSomeField() {
        return this.someField;
    }

}
```

¿En qué consiste?

Es una interfaz para manipular fechas y horas, reemplazando las APIs de Java antiguas de las clases *Date* y *Calendar*. Sus principales ventajas con respecto a la antigua API son: la inmutabilidad, la seguridad en hilos concurrentes y el manejo de husos horarios.

Clases Principales

1. **LocalTime**: Representa una hora.
2. **LocalDate**: Representa una fecha.
3. **LocalDateTime**: Denota una fecha y una hora.
4. **ZonedDateTime**: Agrega el huso horario dentro de la representación.
5. **Period**: Abarca un espacio de tiempo, como los años o los días. Como ejemplo, se puede utilizar con las fechas para determinar rangos.
6. **Duration**: Especifica las duraciones de tiempo en nanosegundos, y se puede representar como segundos, horas, etc. Como ejemplo, esta clase se puede utilizar para extraer rangos de tiempo a partir de las clases que ofrecen una hora.

Cada una de estas clases contienen un abanico de operaciones y constructores para trabajar con fechas y horas. Se pueden crear fechas a partir de texto:

```
LocalDate.parse("2015-02-20");
```

Incluso se pueden añadir o restar días a la fecha resultante:

```
LocalTime.parse("06:30").plus(1, ChronoUnit.HOURS);
```

Características

- **Inmutable:** No se puede modificar una instancia de estas clases. El método *with* sirve como un "modificador" de una hora o fecha, devolviendo una instancia nueva con los nuevos valores deseados.
- **Seguridad de hilos:** Como consecuencia de la inmutabilidad, instancias de esta API se pueden utilizar dentro de hilos sin preocuparse de problemas de concurrencia que podrían ocurrir con instancias mutables.
- **Husos horarios:** La implementación previa de fechas y horas no ofrecían el manejo de husos horarios. El programador tenía que buscarse o implementarse uno. La nueva API ofrece esta capacidad de manejar horas y fechas con la clase *ZonedDateTime*.
- **Métodos consistentes:** Cada clase tiene un grupo de métodos con los mismos nombres, facilitando su uso. Por ejemplo, el método *of* se especifica en cada clase como la manera de inicializar un objeto de esa clase.
- **Estandarizado:** El diseño de la API se centra en el estándar ISO 8601 para fechas y horas.

¿Qué es?

La **computación concurrente** es una forma de computación en la que varias tareas se ejecutan sin esperar a que la anterior haya terminado, aunque no necesariamente tienen que estar ejecutándose en el mismo instante.

SECUENCIAL, CONCURRENTE Y EN PARALELO

La ejecución de un programa suele realizarse siempre de manera **secuencial**. Si tenemos que realizar las tareas A y B, primero se realiza por completo la tarea A y cuando ha acabado se realiza la tarea B.

Hay ocasiones en las que no nos importa el orden en que se ejecuten las tareas. Esto puede hacerse usando **concurrencia**.

Un procesador mononúcleo solo puede ejecutar una operación a la vez, de manera que no se pueden ejecutar las tareas A y B a la vez. Si estamos usando concurrencia, nuestro procesador puede decidir ejecutar parcialmente la tarea A, luego cambiar y ejecutar la tarea B e ir ejecutando partes de estas tareas de manera intercalada. De esta manera conseguimos que las tareas A y B se estén ejecutando durante el mismo periodo de tiempo a pesar de que en cada instante de tiempo sólo se esté realizando una tarea.

Si tenemos más de un procesador o un procesador multinúcleo podemos ejecutar la tarea A en un núcleo y la tarea B en otro diferente de manera que ambas tareas se ejecutan en el mismo instante en **paralelo.**

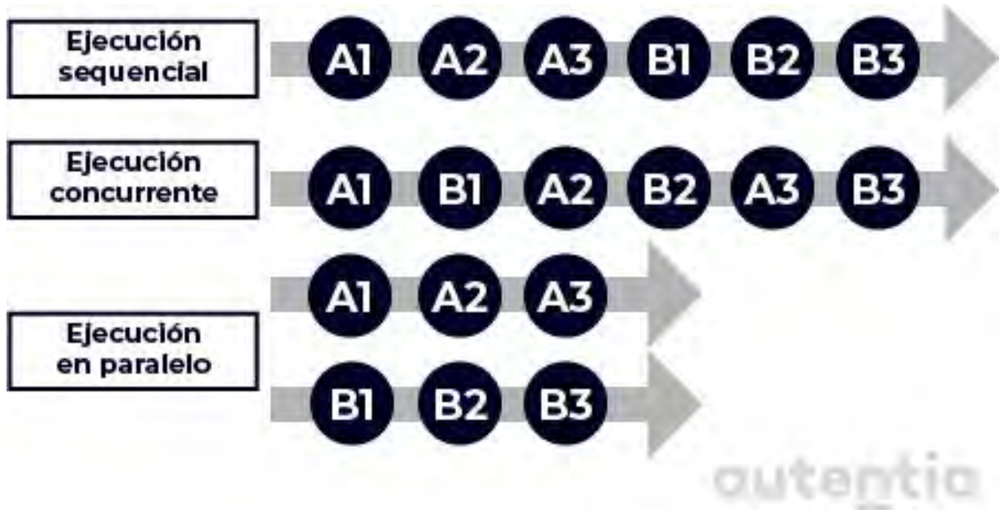
VENTAJAS

- Aumento de la **eficiencia**.
- Disminución de **tiempos de espera.**
- Disminución de **tiempos de respuesta**.

DESVENTAJAS

Al tener que gestionar la concurrencia, nuestro código será más complejo y más difícil de escribir y de mantener. Además, necesitamos tener en cuenta que el código ya no se ejecuta de manera secuencial y, por lo tanto, tenemos que tener especial cuidado con los recursos compartidos. Podemos tener, entre otros, condiciones de carrera, interbloqueos o inanición de procesos.

- **Condiciones de carrera:** Se dan cuando dos o más hilos acceden a un mismo recurso e intentan modificarlo a la vez. El resultado dependerá del orden en que se ejecuten los hilos.
- **Interbloqueo:** Se da cuando un hilo quiere acceder a un recurso que está siendo bloqueado por otro hilo y entra en estado de espera. Si ese otro hilo también está en estado de espera ambos hilos nunca saldrán del bloqueo..
- **Inanición:** Se da cuando un hilo tiene prioridad baja y nunca llega a ejecutarse porque se están ejecutando hilos con más prioridad.



¿Qué son?

Son expresiones que actúan como **funciones anónimas**. Pueden incluirse en cualquier lugar que acepte una **interfaz funcional**.

SINTAXIS

```
(Tipo1 param1, TipoN paramN) -> {cuerpo de la lambda}
```

Sin argumentos (paréntesis obligatorios)	`() -> System.out.println("Hola mundo!")`
1 argumento (paréntesis opcionales)	`cadena -> System.out.println(cadena)`
2 o más argumentos (paréntesis obligatorios)	`(x, y) -> x + y`
Con tipos explícitos (no es necesario incluir tipos)	`(int x, int y) -> x + y`

Con múltiples sentencias
(se utilizan las llaves y return)
	`(x, y) -> {`
	`  System.out.println(x);`
	`  System.out.println(y);`
	`  return x + y;`
	`}`

INTERFACES FUNCIONALES

En Java, se considera interfaz funcional a toda interfaz que contenga **un único método abstracto**. Es decir, interfaces que tienen métodos estáticos o por defecto (default) seguirán siendo funcionales si solo tienen un único método abstracto.

¿DÓNDE SE USAN?

Donde se acepten **interfaces funcionales**. Por ejemplo:

- Al retornar:

```
private Predicate<Integer> isOddPredicate() {
    return n -> n % 2 != 0;
}
```

- En variables:

```
Predicate<Integer> isOdd = n -> n % 2 != 0;
```

- En llamadas a métodos:

```
IntStream.range(1, 11)
    .mapToObj(i -> String.format("i = %s", i))
    .forEach(System.out::println);
```

REFERENCIAS A MÉTODOS

Si la firma de la **interfaz funcional** coincide con la firma de otro método cualquiera, podemos usar una referencia al método en vez de una expresión lambda.

```
System.out::println
this::miMetodo
```

```
super::metodoDeSuper
unObjeto::suMetodo
```

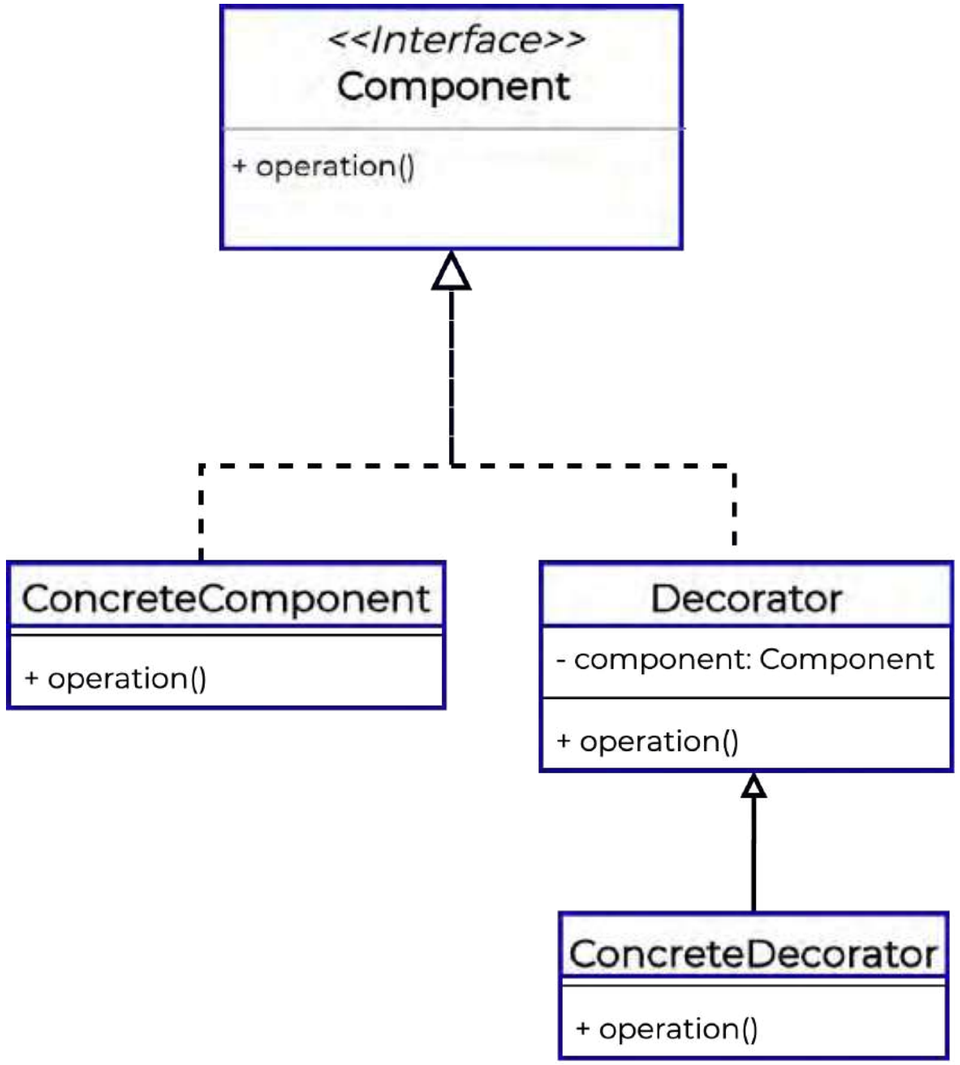
¿En qué consiste?

Patrón que **permite añadir nuevas funcionalidades a un objeto en tiempo de ejecución sin modificar su estructura** y a través de una envoltura (wrapper). El decorador envuelve la clase original sin cambiar la firma de los métodos existentes.

CONCEPTO

Decorator ofrece una alternativa cuando no es posible extender el comportamiento de un objeto a través de la herencia.

Normalmente tenemos una interfaz con varias implementaciones. Para aplicar este patrón, debemos crear una nueva 'implementación' que será nuestro *Decorator*. A partir de esta clase, creamos clases concretas de Decorator con las nuevas funcionalidades que se desean añadir.



Herramientas y técnicas

Sistema de control de versiones distribuido

autentia

¿Qué es?

En inglés **Distributed Version Control System (DVCS)**, es un sistema que permite a los usuarios trabajar en un proyecto común de forma independiente con una copia del repositorio en su máquina local de forma que no se necesita conexión a internet para realizar cambios.

¿EN QUÉ CONSISTE?

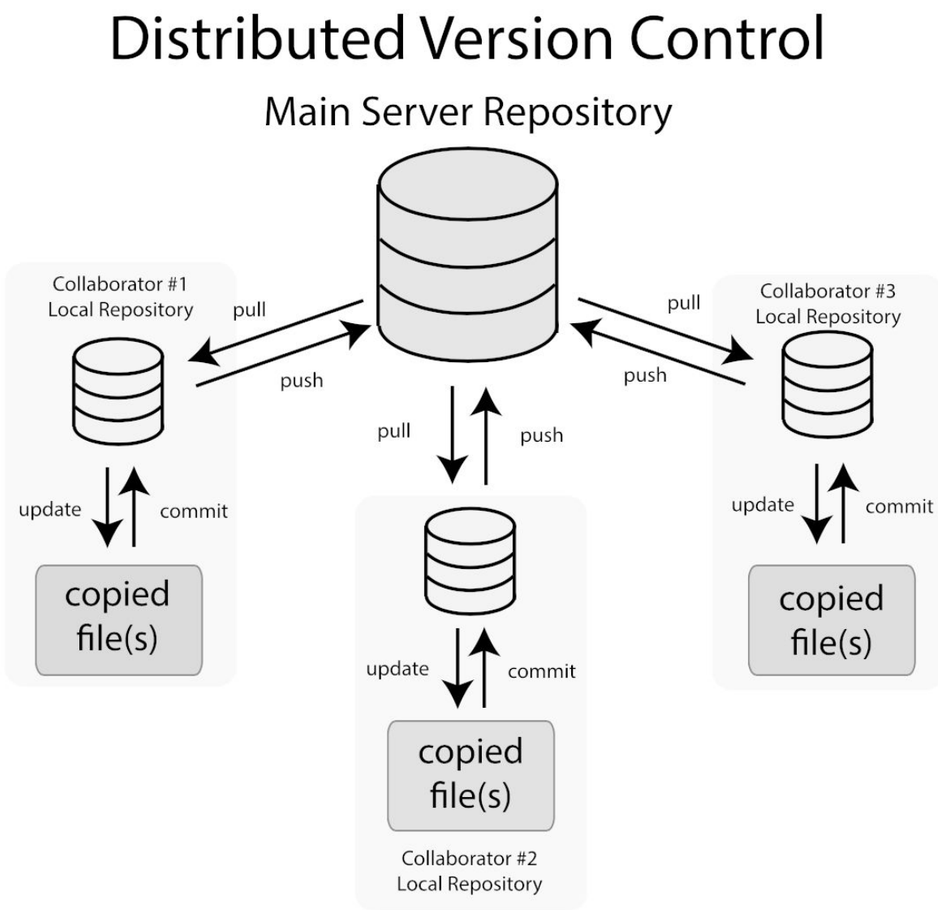
Antiguamente teníamos un repositorio central que estaba situado en una máquina concreta y contenía todo el histórico, etiquetas y ramas del proyecto. En caso de querer hacer un commit, debíamos tener acceso a internet obligatoriamente. Con los **sistemas distribuidos** tenemos la ventaja de **tener una copia del repositorio en local** por lo que no se necesita tener acceso a internet para ir haciendo commits, pudiendo restaurar el repositorio central a partir de la copia local si éste se cae o se elimina. Git sigue este sistema de control de versiones. Algunas preguntas habituales son:

¿Si tenemos todo el repositorio en local, no ocupará mucho espacio?

Lo normal es que no, porque al ser un repositorio distribuido, sólo tendremos la/las ramas que nos interesen.

¿Si todo el mundo trabaja en local, no puede resultar que distintas ramas diverjan o entren en conflicto?

Sí, y esto es normal que suceda. Todo al final depende de nuestro proceso de desarrollo, no tanto de la herramienta que usemos; es normal que los desarrolladores abran nuevas ramas constantemente para desarrollar nuevas funcionalidades y que luego, se haga un merge para unificar estos cambios sobre otra rama.



Fuente: https://code.snipcademy.com/tutorials/git/introduction/how-version-control-works

¿Qué es?

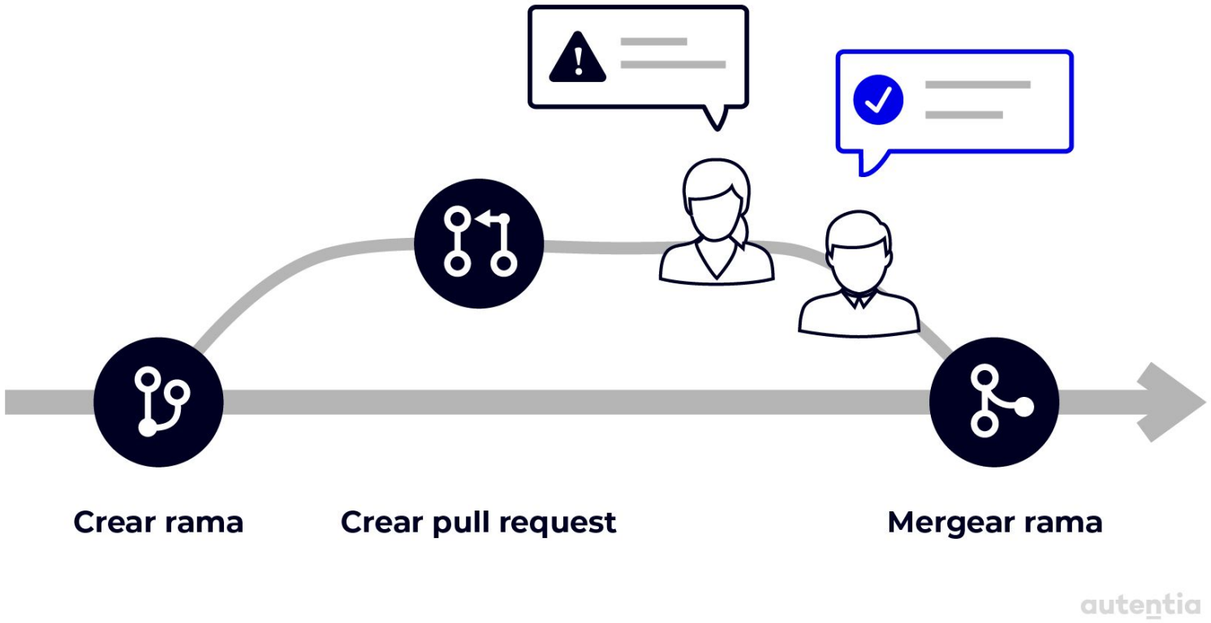
Un Pull Request es una **solicitud que realiza un desarrollador ante un cambio de código fuente** realizado en un proyecto software **para fusionarlo con otra rama**, habitualmente la rama principal.

BENEFICIOS DEL PULL REQUEST

- El Pull Request es un muy buen **procedimiento de Team Building** dentro de tu equipo ya que va a permitir la mejora del mismo mediante las revisiones de código donde los miembros pueden dar un feedback valiosísimo.

- Que se realicen Pull Request va a permitir **encontrar incidencias y defectos en el código en etapas más tempranas** lo que va a suponer un ahorro para la empresa.

- Como equipo el Pull Request nos **hace ser responsables del commit,** lo que va a forzar que se hagan revisiones de código, dar feedback e información que conlleva ese commit.



¿Cómo gestionarlas?

A la hora de trabajar con un sistema de control de versiones y en un equipo multidisciplinar, es necesario definir el flujo de trabajo que va a seguirse. GitFlow es una de las estrategias para la sincronización de ramas más comunes, fusionando los cambios introducidos por el equipo a través de Merge/Pull requests.

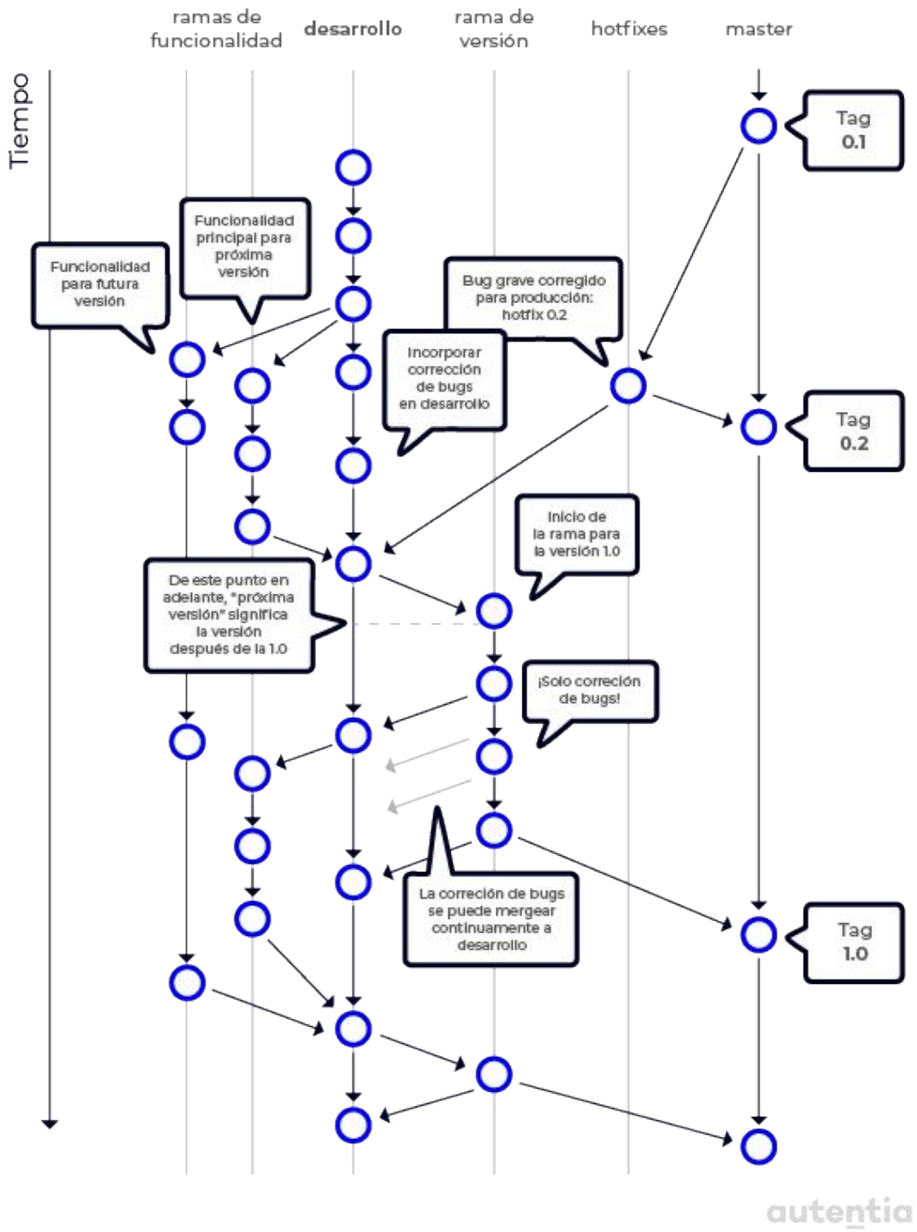
GIT FLOW

Modelo que establece una política de ramas bastante adaptable a cualquier entorno aunque puede llegar a ser compleja. Tiene una rama de desarrollo principal que suele llamarse **develop** y a partir de ella se crean subramas llamadas **features** para desarrollar nuevas funcionalidades. La rama **release** es una rama previa a producción que si finalmente todo está bien, se hace merge a **master**.

¿Qué pasa si encontramos un bug en producción? las ramas **hotfix** nacen a partir de master para solucionar problemas de urgencia y posteriormente se hace el merge a master y develop. **Los únicos commits que podemos encontrar en master son los merge con la rama release y hotfix**.

PULL REQUEST Y MERGE REQUEST

Previamente a hacer un merge, podemos realizar lo que se conoce como pull request en Github o merge request en Gitlab. Es un mecanismo que permite controlar los cambios de código introducidos en ciertas ramas, mediante un sencillo proceso de aprobación. Los motivos de rechazo pueden ser muy variados, desde conflictos en el código hasta desechar completamente una funcionalidad.



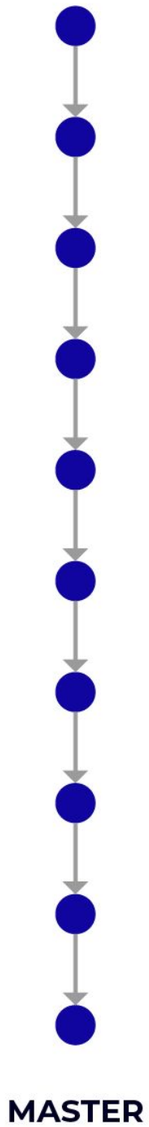
¿Cómo gestionarlas?

A la hora de trabajar con un sistema de control de versiones y en un equipo multidisciplinar, es necesario definir el flujo de trabajo que va a seguirse. Trunk based development es la estrategia utilizada cuando se opta por una entrega continua, integrando pequeñas funcionalidades de manera frecuente.

TRUNK BASED DEVELOPMENT

Modelo de desarrollo muy simple pero también usado de forma diferente según el tamaño del equipo. La idea general es que **todos los desarrolladores trabajen en una sola rama**, normalmente suele ser la rama maestra (master), también conocida como *trunk*. Se suele usar este modelo cuando se necesita iterar rápido y el equipo de desarrollo está en continua comunicación.

TIene la ventaja de que se **integran los cambios de forma muy rápida** ya que obliga al equipo a realizar pequeños commits e ir haciendo push con mucha frecuencia. Además, cada persona verifica que todo los tests, incluidos los de integración, pasan sin ningún problema. Otra ventaja es que el tamaño de los merge suele ser muy pequeño (se evita el *merge hell* de cientos de ficheros) y por consiguiente se reducen los conflictos. Es un metodología opuesta a Git Flow.



OTROS MODELOS

A parte de los dos modelos vistos en estas dos fichas, existen otros modelos de gestión también conocidos como **Github Flow** que solo tiene dos ramas master y las features y se van haciendo pull request para hacer el merge a master, **Gitlab Flow, Release Flow** por Microsoft, entre otros.

¿Qué es?

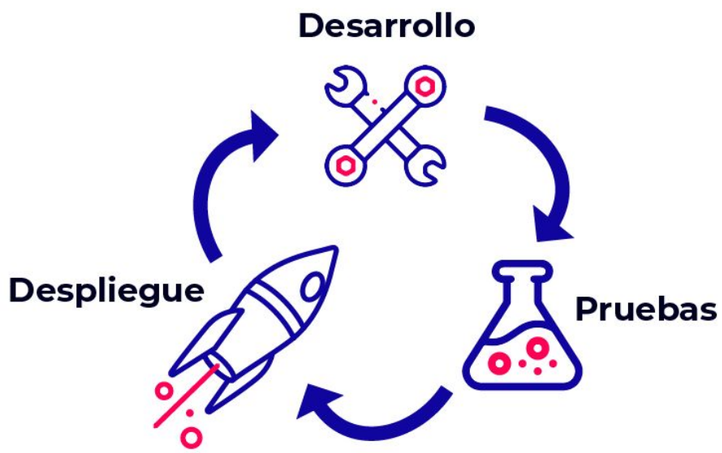
En inglés **Continuous Integration (CI)** es la práctica que tiene como objetivo integrar los cambios en el repositorio central de forma periódica a través de varios de procesos automatizados donde cada versión generada se comprueba mediante pruebas y tests para detectar posibles errores de forma temprana.

¿EN QUÉ CONSISTE?

Durante el proceso de integración continua se pasan ciertas fases o tareas a ejecutar, como por ejemplo, instalar las dependencias necesarias, lanzar los tests, entre otras. En caso de que alguien del equipo decida hacer un Pull Request/Merge Request, comprobamos que dicho pipeline ha pasado correctamente y procedemos a integrar dichos cambios ya sea en la rama principal o en nuestra propia rama local en caso de necesitarlos.

¿Qué beneficios nos aporta?

- **Detección rápida de fallos** de forma continua.
- **Aumento de la productividad del equipo**.
- **Automatización** y ejecución inmediata de procesos.
- **Monitorización** continua de las métricas de calidad del proyecto.



Debemos tener en cuenta:

- **No dejar pasar más de dos horas sin integrar los cambios** que hemos programado.
- La programación en equipo es un problema de **“divide, vencerás e integrarás”.**
- **La integración es un paso no predecible** que puede costar más que el propio desarrollo.
- **Integración síncrona:** cada pareja después de un par de horas sube sus cambios y espera a que se complete el build y se hayan pasado todas las pruebas sin ningún problema de regresión.
- **Integración asíncrona**: cada noche se hace un build diario en el que se construye la nueva versión del sistema. Si se producen errores se notifica con alertas de emails.
- **El sistema resultante debe ser un sistema listo para lanzarse**.

¿Qué es?

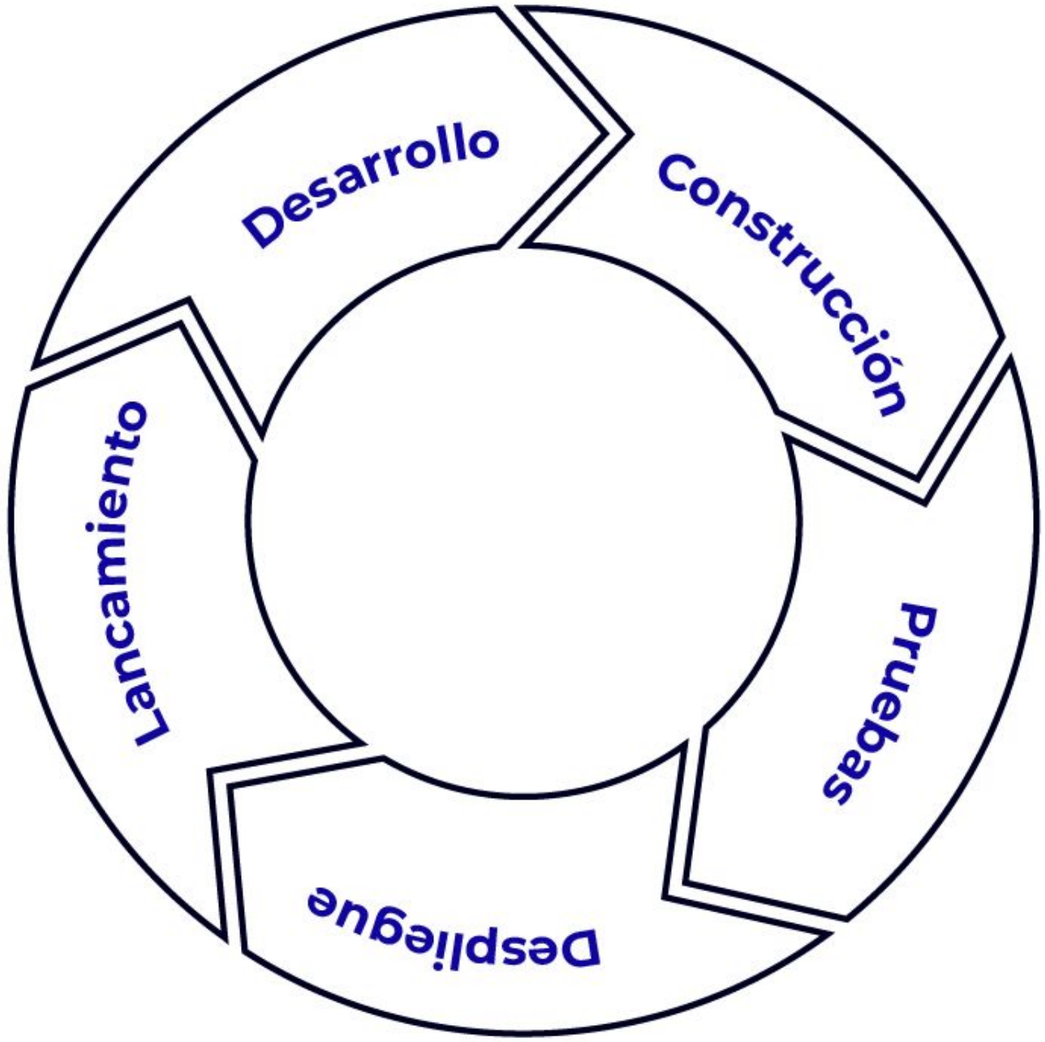
En inglés **Continuous Deployment (CD),** es una práctica que tiene como objetivo proporcionar una manera ágil, fiable y **automática** de poder entregar o desplegar los nuevos cambios en el entorno específico, normalmente producción. Suele utilizarse junto con la integración contínua.

¿EN QUÉ CONSISTE?

El despliegue continuo se basa en automatizar todo el proceso de despliegue de la aplicación en cualquiera de los entornos disponibles, ya sea sólo en algunos de ellos o en todos, todo ello, sin que haya **ninguna intervención humana** en el procedimiento.

El objetivo es hacer **despliegues predecibles, automáticos y rutinarios** en cualquier momento, ya sea de un sistema distribuido a gran escala, un entorno de producción complejo, un sistema integrado o una aplicación.

Dependiendo de cada proyecto o propósito de negocio de la empresa, el despliegue estará configurado para hacerse de forma semanal, quincenal o cada 2 o 3 días, aunque el objetivo es hacerlo de manera constante y en periodos cortos para obtener feedback rápido del cliente.



¿Qué es y para qué sirve?

En inglés **Versioning**. Cada release generada de un artefacto software debe estar etiquetada a través de un número de versión. Dicho número de versión debe seguir un formato específico para así cada release ser identificada de manera única y aportar información de su naturaleza y objetivo (nuevas features, corrección de bugs, evolutivos...). El esquema comúnmente adoptado es el indicado en la especificación *Semantic Version (SemVer).*

RELEASE VERSION X.Y.Z

- **X** representa el número de versión **MAJOR.**
- **Y** representa el número de versión **MINOR.**
- **Z** representa el número de versión **PATCH.**

Incremento del número de versión **MAJOR, MINOR o PATCH** por cada nueva release del artefacto software en función de la naturaleza del cambio. Dada una release con número de versión X.Y.Z y una nueva versión a partir de ella:

- **Incrementar X (MAJOR version)** si la nueva versión incluye cambios de API incompatibles.
- **Incrementar Y (MINOR version)** para evolutivos o cambios retrocompatibles con la release X.Y.Z.
- **Incrementar Z (PATCH version)** para correcciones de bugs de la release X.Y.Z.

A TENER EN CUENTA

1. Si se **incrementa la versión MAJOR se resetean los valores de MINOR y PATCH** (p. ej. La nueva major de la release 3.2.1 sería la 4.0.0).
2. Si se **incrementa la versión MINOR se resetea el valor de PATCH** (p. ej. La nueva minor de la release 3.2.1 sería la 3.3.0).
3. Si se **incrementa la versión PATCH los valores de MAJOR y MINOR no se modifican** (p. ej. Un hotfix sobre la release 3.2.1 genera una nueva release con versión 3.2.2).
4. El **incremento de MAJOR, MINOR y PATCH es secuencial**.
5. Los **cambios sobre una release generada** deben hacerse sobre **una nueva versión.**
6. Cada **release debe ser identificada de manera única.**
7. Pueden incluirse nuevos tags en las versiones de releases actualmente en desarrollo (3.0.0-alpha, 3.0.0-SNAPSHOT, 1.0.0-0.3.7).
8. La precedencia de las releases viene determinado por el valor de MAJOR, MINOR y PATCH y pre-release en este orden (0.2.0 < 0.2.1 < 0.3.1 < 1.0.0-SNAPSHOT < 1.0.0).

¿Qué son?

Son las principales herramientas para la gestión de proyectos: compilación, tests, gestión de dependencias, integración con herramientas de integración continua, despliegue o generación de releases.

MAVEN

- Configuración del proyecto a través de ficheros XML (pom.xml).
- Gestión de dependencias clara y sencilla.
- Definición de repositorios de donde descargar las dependencias del proyecto.
- Ciclo de vida basado en fases predefinidas. Cada fase contiene goals (p. ej. Empaquetado, ejecución de tests...).
- Es la herramienta de gestión de configuración más utilizada en el desarrollo de proyectos software.
- Amplio catálogo de plugins desarrollados.
- Gran soporte por parte de la comunidad.
- Soporta la integración con herramientas de integración continua como Jenkins o Travis.

GRADLE

- Configuración del proyecto a través de ficheros escritos en Groovy o Kotlin (build.gradle).
- Configuración basada en proyectos y tareas. Un proyecto representa qué se quiere hacer y está compuesto de varias tareas.
- Como se tiende a programar tareas propias, los scripts pueden variar de un proyecto a otro.
- Gestión de dependencias.
- Soporta la integración con herramientas de integración continua como Jenkins o Travis.
- Integración con otras herramientas de gestión como Maven o Ant.

¿Qué es?

Quality assurance (QA) es la manera de medir la calidad de software con un conjunto de actividades como son estrategias de pruebas con distintos enfoques (de aceptación, de carga...), uso de estándares y monitorizar el producto para detectar el impacto de los cambios entre otras. Actividades que intentan asegurar al cliente que se está aportando un producto que cumple con sus expectativas con la máxima calidad y el menor número de errores posible.

CONCEPTO

El equipo de QA debe ser consciente de las expectativas que se persiguen, tener claro los requisitos y la idea que tiene el Product Owner, teniendo una **comunicación constante con el equipo de negocio y el de desarrollo.** Su función no es corregir el código de los desarrolladores evitando que estos hagan pruebas. Todo lo contrario, QA proporciona un filtro extra que abarca un mayor campo de test como son los tests de de aceptación, poniendo el sistema a prueba para cada uno de los casos de uso definidos.

Otra de las tareas importantes de QA es **preparar un entorno de pruebas similar al de producción,** usando datos reales en las pruebas.

La realización de estas pruebas tampoco aseguran al 100% que no se vayan a producir errores en fases posteriores, pero sí de que se reduzcan y que los que aparezcan tengan un menor impacto económico al haber cubierto la mayor parte de los casos de usos por tests.

Fuente: https://www.techentice.com/the-future-of-quality-assurance-qa/

Given - When - Then

autentia

¿Qué es?

Given - When - Then es un patrón para escribir las pruebas mediante la especificación del comportamiento de un sistema (Specification By Example). Es un enfoque desarrollado por David Terhorst y Chris Matts como parte de BDD (Behavior-Driven Development).

¿CUÁL ES EL ENFOQUE?

El enfoque consiste en que los requisitos sobre el comportamiento nos vienen de negocio y somos los desarrolladores, junto con negocio los que preguntamos el por qué de la funcionalidad, extraemos el comportamiento dados unos escenarios y enumeramos las posibles respuestas dada una acción.

El patrón es muy simple, para cada caso de prueba se utiliza la siguiente sintaxis:

GIVEN [Escenario]: especificamos los parámetros de la función.

WHEN [Acción]: indicamos la acción realizada.

THEN [Resultado]: formulamos el resultado esperado.

```
testMethod(){
GIVEN
... <parámetros o entrada de datos>
WHEN
... <acción realizada>
THEN
... <resultado esperado>
}
```

EJEMPLO

Se da el hecho de que en este enfoque es muy común utilizar ANDs en cada cláusula para combinar múltiples expresiones.

```
Feature: Pago de compra online mediante tarjeta de crédito

Como usuario registrado que realiza una compra online
Quiero poder realizar el pago mediante tarjeta de crédito
Para facilitar el pago aplazado

Scenario: Usuario registrado solicita el pago de una compra
    Given que he accedido a la sección de pago con tarjeta de
crédito
      And dispongo de un crédito de 1200 euros en la tarjeta
      And la fecha de caducidad de la tarjeta no está cumplida

    When relleno el campo nombre con "Paco"
      And relleno el campo apellidos con "López García"
      And relleno el campo tarjeta con "1234 5678 9012 3456"
      And relleno el campo CCV con "123"
      And relleno el campo F.caducidad con "20/05/2025"
      And hago click en pagar.
    Then se debería abrir una ventana nueva con la pasarela de
pagos
      And deberíamos ver un mensaje de datos verificados
      And deberíamos recibir un email con el justificante de pago
```

El mundo de los microservicios

¿Qué es?

Es el framework más popular para el desarrollo de aplicaciones en Java. El ecosistema de Spring es muy extenso, incluyendo 24 proyectos, de los cuales los más conocidos son **Spring Boot** y **Spring Data**.

VENTAJAS

- **Madurez:** lleva ya casi 20 años en el mercado.
- **Configuración:** la configuración por defecto de Spring cubre las necesidades de la mayoría de desarrollos
- **Productividad:** nos permite hacer código limpio y fácilmente testeable, haciendo que los desarrollos sean rápidos, robustos y fáciles de extender.
- **Velocidad:** optimizado para tener unos tiempos de compilación y ejecución muy rápidos y con cada nueva versión se mejoran.
- **Seguridad:** viene con la seguridad integrada, de manera que tu aplicación cumplirá con los estándares más recientes de seguridad sin tener que configurar nada.

SPRING FRAMEWORK

Las funcionalidades y configuraciones más fundamentales son:

- **Inyección de dependencias:** permite usar fácilmente este paradigma.
- **Programación orientada a aspectos:** puedes implementar las funcionalidades transversales como la seguridad tan fácil como poner una anotación resultando en un código limpio y fácil de mantener.
- **Testing:** al usar inyección de dependencias es más sencillo hacer tests unitarios. Además Spring nos ofrece multitud de utilidades para facilitar tests de integración.
- **Acceso a datos:** proporciona una API consistente para usar JTA, JDBC, Hibernate y JPA.
- **Spring MVC:** se usa para crear aplicaciones web.

MÓDULOS DE SPRING

Spring tiene más de 20 proyectos, entre los cuales están:

- **Spring Boot:** permite crear aplicaciones de spring de manera muy rápida.
- **Spring Data:** proporciona un modelo consistente para acceder a bases de datos relacionales y no relacionales, servicios de datos en la nube, map-reduce y más.
- **Spring Cloud:** proporciona herramientas para funcionalidades comunes usadas en sistemas distribuidos. Es útil para crear y desplegar microservicios.

Inversión de control

autentia

¿Qué es?

En la programación tradicional, la interacción entre clases y funciones se hace de forma imperativa. La inversión de control es un principio que delega en un tercero (un framework o contenedor) el control del flujo de un programa para la creación de un objeto, la inyección de objetos dependientes, etc.

¿EN QUÉ CONSISTE?

La inversión de control **está basada en el principio de Hollywood**, ya que era muy habitual la frase que decían los directores a los aspirantes:

"No nos llames; nosotros te llamaremos".

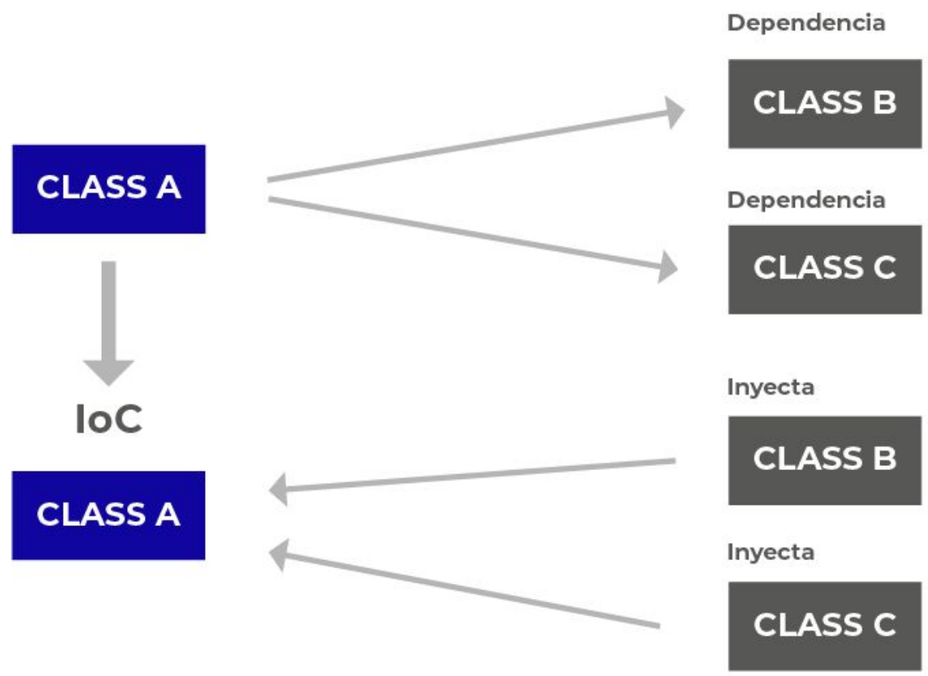
Podemos encontrar distintas formas de implementar la inversión de control. Las formas más conocidas de este principio son:

- Localizador de servicios (Service Locator).
- Inyección de dependencias.

La diferencia fundamental entre ambas es que con el Service Locator las dependencias aún se solicitan **explícitamente** desde la clase dependiente, mientras que con la inyección de dependencias, un agente externo se encarga de proveer las mismas, sin mediar acoplamiento entre la dependencia y su proveedor.

VENTAJAS

- **Reduce el acoplamiento** entre clases y a su vez **aumenta la modularidad y extensibilidad**.
- A raíz del punto anterior, **mejora la testeabilidad** del código ya que al reducir el acoplamiento podemos crear dobles de prueba de las dependencias de una clase de una forma muy sencilla.



¿Qué son?

Las anotaciones son una característica de Java que nos permite asociar un elemento de nuestro código a una serie de metadatos. Estos metadatos son utilizados por el compilador o durante la ejecución del programa, donde otros frameworks y librerías se pueden aprovechar para generar código o realizar otras operaciones.

¿CÓMO DECLARAR UNA ANOTACIÓN?

Una anotación se declara con **@** seguido por su nombre. **Debe preceder el elemento que queremos anotar**. Por ejemplo, un caso sencillo es la anotación **@Override**, indicando que el elemento siguiente sobreescribirá un elemento de la superclase donde está definida.

Algunas anotaciones pueden tener elementos asignables. En este caso se definen entre paréntesis después del nombre de la anotación:

@Author(nombre="José", apellido="Palacios")

Otras características:

- Se pueden declarar varias anotaciones para un mismo elemento.
- Normalmente, una anotación ocupará su propia línea, pero es meramente, una convención de estilo.

```
@Override
@Author(nombre="Juan", apellido="Palacios")
private String title;
```

EJEMPLO

Lombok es una librería que ofrece anotaciones para generar código. Una de sus anotaciones, @Getter, se asocia a una variable dentro de una clase:

```
public class LombokExample {

    @Getter
    private int someField;

 }
```

Al compilarla, se añadiría un método a la clase compilada:

```
public class LombokExample {

    private int someField;

    public int getSomeField() {
        return this.someField;
    }

}
```

Un único ser sin igual

El patrón *Singleton* resuelve el problema de mantener una única instancia de una clase en memoria durante la ejecución del programa.

DISEÑO

El diseño de este patrón impide que otras clases creen nuevas instancias del Singleton. La primera vez que una clase la necesite, se creará la primera y única instancia de ella.

A partir de ahora, cada vez que se solicita una instancia del Singleton, se hará referencia a la misma instancia, asegurándonos de que no se cree otra.

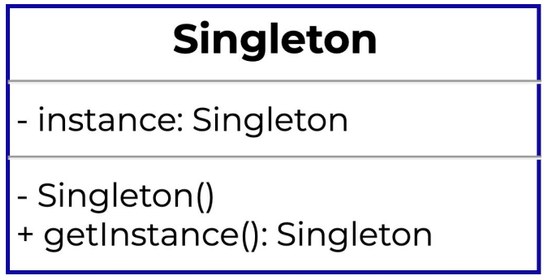
PRECAUCIÓN

Este patrón se comporta como un objeto global, donde cualquier parte de la aplicación la puede utilizar.

Cambios hechos al estado de una instancia Singleton pueden repercutir en otras clases que la utilicen. Si ocurriese algún problema, es probable que nos resulte difícil de detectar y corregir.

También dificulta el desarrollo de pruebas, ya que el uso de un Singleton implica una dependencia oculta que al momento de hacer pruebas, puede causar sorpresas.

APLICACIÓN



A partir del diagrama se infiere que no se podrán crear nuevas instancias de Singleton, dado que el constructor es privado. El método *getInstance* debe ser estático y será el punto de acceso para utilizar la referencia encontrada en *instance*.

UTILIDAD

Un uso común de este patrón se encuentra en componentes que quieren restringir su uso desde un solo punto:

- **Parámetros de entorno o de configuración:** permite tener una fuente única y rápida de información. Sólo lectura.
- **Acceso a interfaces de hardware:** se restringe el acceso a recursos que deben ser utilizados uno a la vez y no se pueden paralelizar.

Domain Driven Design

autentia

¿Qué es?

El diseño guiado por el dominio es un enfoque para el desarrollo de software con necesidades complejas mediante una profunda conexión entre la implementación y los conceptos del modelo y núcleo del negocio. El término fue acuñado por Eric Evans en su libro *"Domain-Driven Design - Tackling Complexity in the Heart of Software".*

¿QUÉ PROBLEMA INTENTA RESOLVER DDD?

- Depende del caso o **proyecto** nos podemos encontrar con que la **complejidad** de muchas aplicaciones no está en la parte técnica sino **en la lógica del negocio o dominio.**

- El dilema empieza cuando intentamos **resolver problemas del dominio con tecnología**. Eso provoca que, aunque la aplicación funcione, no haya *nadie capaz de entender realmente cómo lo hace*. Es habitual que surja el anti-patrón *"Modelo del Dominio Anémico"* (en inglés Anemic Domain Model).

¿CÓMO INTENTA RESOLVERLO?

- El **DDD no es una tecnología ni una metodología**, éste *provee una estructura de prácticas y terminologías* para tomar decisiones de diseño que enfoquen y aceleren el manejo de dominios complejos en los proyectos de software.

- **Proporciona una serie de patrones tácticos y estratégicos** que nos ayudan a trabajar con los expertos del dominio modelando el problema y la solución. De este modo, se inicia una colaboración creativa entre técnicos y expertos de dominio para interactuar lo más cercano posible a los conceptos fundamentales del problema.

¿QUÉ CONCEPTOS CLAVE UTILIZA?

- El **lenguaje ubicuo** (lenguaje común entre los programadores y los usuarios) y el **bounded context** (identificación de los límites de los diferentes dominios/subdominios) principalmente.

¿QUÉ CARACTERÍSTICAS DEBE TENER UN PROYECTO PARA QUE SEA INTERESANTE APLICAR DDD?

- Tenemos un **dominio complejo**.
- No tenemos ni idea del dominio, pero **sabemos que vamos a tener muchos procesos, HdUs, etc.**
- Se trata de un proyecto con **proyección a varios años vista.**

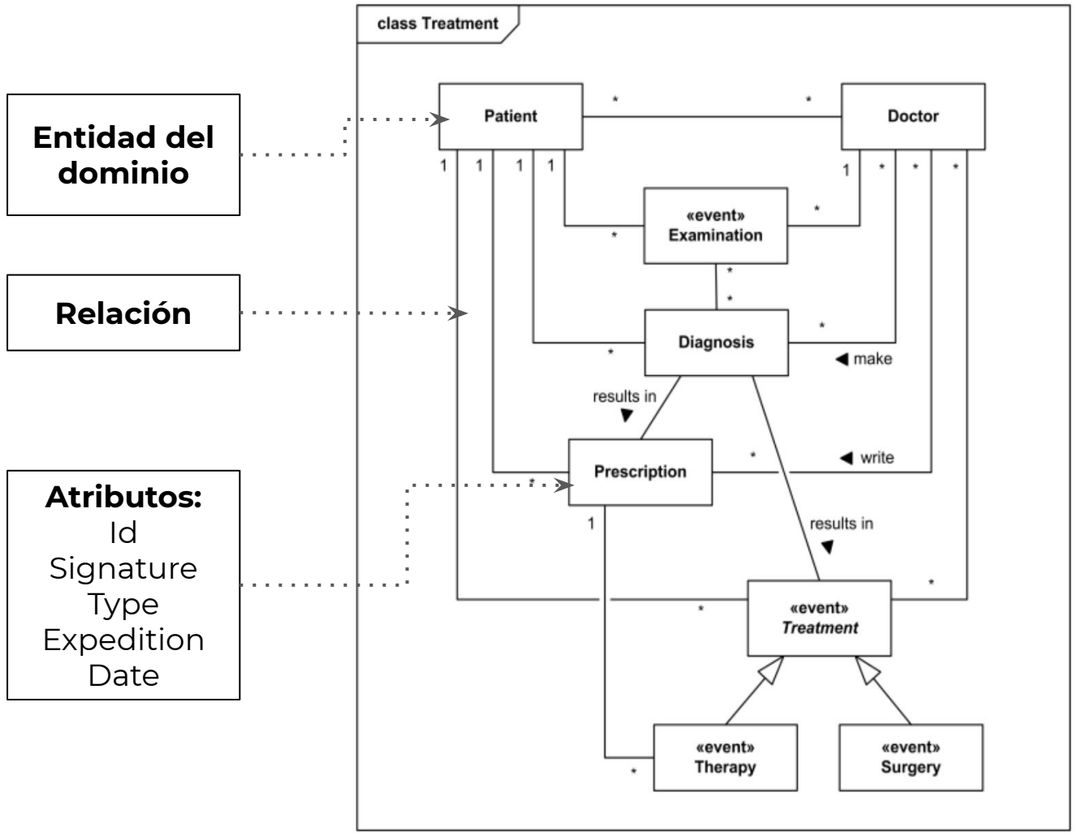
¿QUÉ PRE-REQUISITOS SE NECESITAN PARA APLICAR DDD?

- El **desarrollo debe ser iterativo.** Esto será necesario para ir refinando el modelo del dominio continuamente a medida que aprendemos más sobre este y avanzamos.

- **Debe existir una estrecha relación entre los desarrolladores y los expertos del dominio.** El conocimiento profundo del dominio es esencial, al igual que la colaboración con los expertos de desarrollo durante la vida del proyecto; esto evitará malos entendidos entre las partes del equipo y ofrecerá la oportunidad de obtener un conocimiento más profundo del dominio.

EL MODELO DE DOMINIO

- Es un **modelo conceptual de todos los temas relacionados con un problema en específico**.
- Formalmente en él **se representan, mediante dibujo y texto,** las distintas entidades, sus atributos, papeles y relaciones, así como las restricciones que rigen el dominio del problema.
- Su **finalidad es representar el vocabulario y los conceptos clave del dominio del problema.**



DOMAIN STORYTELLING

- La mejor forma de aprender un nuevo idioma es escuchar a las personas hablar ese idioma**.** Con el **Domain Storytelling** puedes **emplear el mismo principio** al aprender un nuevo idioma de dominio.
- Deja que los expertos en cada dominio cuenten sus historias de dominio. Mientras escuchas, registra las historias de dominio utilizando un lenguaje pictográfico. **Los expertos en dominios pueden ver de inmediato si entiendes su historia correctamente.** Después de muy pocas historias, serás capaz de hablar sobre las personas, tareas, herramientas, elementos de trabajo y eventos en ese dominio.

EVENT STORMING

- **Event Storming es un método** basado en un taller que **trata de descubrir rápidamente qué está sucediendo en el dominio de un programa de software**. Fue introducido en un blog por *Alberto Brandolini* en 2013.
- **Se trata de discutir** el flujo de eventos de la organización **y de modelar este flujo** de una forma fácil de entender.

Transaccionalidad: ACID

autentia

¿Qué es?

El principio ACID es un estándar de las bases de datos relacionales que se deben cumplir para que se puedan realizar las transacciones en ellas. ACID es el acrónimo en inglés de **A**tomicity, **C**onsistency, **I**solation y **D**urability (Atomicidad, Consistencia, Aislamiento y Durabilidad).

EN DETALLE

Es muy probable que si trabajas o te has movido en torno a las bases de datos en algún momento, hayas oído hablar del término ACID. Este hace referencia a las propiedades que debe cumplir una transacción para que se considere confiable. El término fue ideado por Andreas Reuter y Theo Härde en 1983.

¿Qué entendemos por transacción? En base de datos una transacción consiste en una operación lógica. Por ejemplo, aumentar el IVA de los productos de nuestra base de datos es una única transacción pero puede conllevar acciones sobre numerosas tablas.

Veamos cada uno de los términos:

- **Atomicidad.** Una transacción puede estar compuesta por varias operaciones, si una operación falla todas fallan, por tanto la base de datos se mantiene inmutable.
- **Consistencia.** Asegura que cualquier transacción realizada llevará a la base de datos de un estado válido a otro válido. Por tanto, los datos deben respetar todas las reglas y restricciones definidas.
- **Aislamiento.** Asegura que la ejecución concurrente de varias transacciones deja a la base de datos igual que si estas se hubieran ejecutado de forma secuencial.
- **Durabilidad.** Indica que una vez confirmada una transacción los datos deben persistir en la base de datos

A — **Atomicidad:** las transacciones son todo o nada.

C — **Consistencia:** Sólo se guardan los datos que respetan las reglas definidas.

I — **Aislamiento:** Las transacciones no se afectan las unas a las otras.

D — **Durabilidad:** Los datos que se hayan escrito no se perderán.

¿En qué consisten?

Los verbos HTTP nos permiten realizar distintas operaciones (p. ej. alta, baja, lectura, modificación...) sobre los recursos de nuestras APIs REST. Cada uno de ellos se utiliza para una operación y finalidad concreta.

TIPOS DE VERBOS HTTP

- **GET**: **recuperar** la información de un único recurso o un listado (p. ej. Un listado de cursos, un curso a partir de su id...).
- **POST: dar de alta un recurso**. En el cuerpo de la petición se le proporciona la información a dar de alta.
- **PUT: modificar un recurso existente.** En el cuerpo de la petición se le proporciona la información del recurso actualizada.
- **PATCH:** una **modificación parcial de un recurso.** En el cuerpo de la petición se le proporciona la información a actualizar.
- **DELETE:** dar de **baja un recurso**. En la URL de la petición se le especifica el identificador del recurso a dar de baja.
- **OPTIONS:** se utiliza para describir las opciones de comunicación con el recurso.
- Otros: **HEAD, CONNECT y TRACE.**

CARACTERÍSTICAS

Un verbo HTTP puede ser:

- **Idempotente:** el resultado de la operación deja al servidor en el mismo estado tantas veces se ejecute. GET, PUT, DELETE y HEAD son idempotentes. POST *no* es idempotente.
- **Seguro:** un verbo HTTP es seguro cuando no altera el estado del servidor. GET y OPTIONS son seguros. POST, PUT y DELETE no son seguros. *Todo verbo seguro es idempotente.*
- **Cacheable:** la respuesta a la petición HTTP se guarda en caché de modo que pueda utilizarse en próximas peticiones. GET y HEAD son cacheables mientras que PUT y DELETE *no*, llegando a invalidar resultados cacheados de GET o HEAD.

Estructura para intercambiar información

JSON (JavaScript Object Notation) es un formato para representar datos. Es fácil de interpretar tanto para humanos como para máquinas. Define estructuras comunes encontradas en los lenguajes de programación, facilitando el intercambio de información entre programas.

COMPOSICIÓN

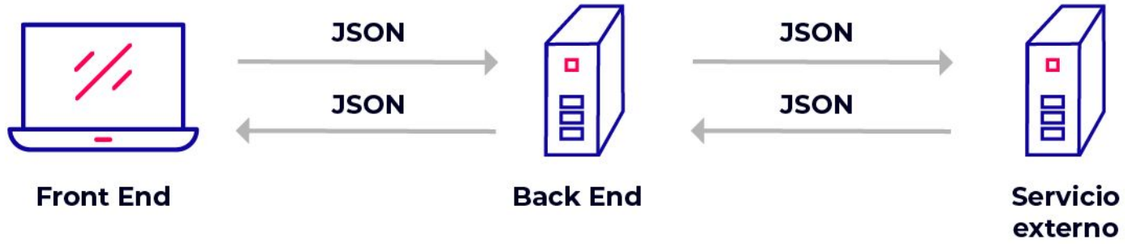
Un objeto JSON se compone utilizando:

1. Pares de clave/valor, como un diccionario.
2. Listas de valores.

Se agrupan estos elementos dentro de objetos, definidos por llaves ('{', '}'), y se separan utilizando comas.

```
{
    "nombre": "valor",
    "objeto": {
        "lista": [
          1, true, "otro_valor"
        ]
    }
}
```

- Las listas, al igual que los valores, son precedidas por una clave.
- Los valores pueden ser números, cadenas de caracteres, otros objetos, listas, etc. Una explicación detallada de la notación se puede conseguir aquí.



A través de llamadas HTTP, se puede transferir información entre distintas aplicaciones utilizando JSON

UTILIDAD

1. Su composición permite agrupar todo tipo de datos que una aplicación pueda necesitar.
2. Son sencillos de utilizar. Interpretadores de JSON han sido implementados por un gran número de lenguajes de programación.

Estructura para intercambiar información

JSON (JavaScript Object Notation) es un formato para representar datos. Es fácil de interpretar tanto para humanos como para máquinas. Define estructuras comunes encontradas en los lenguajes de programación, facilitando el intercambio de información entre programas.

COMPOSICIÓN

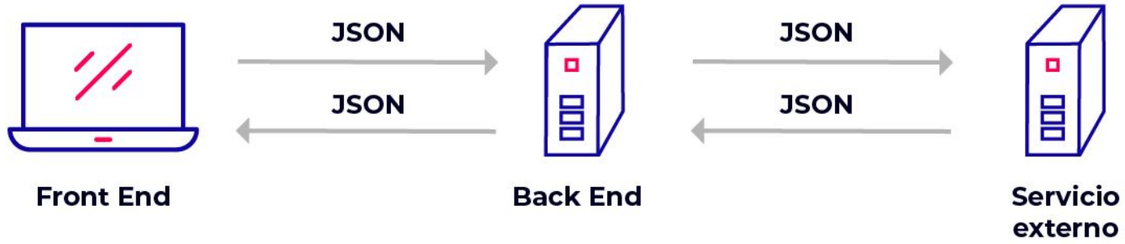
Un objeto JSON se compone utilizando:

1. Pares de clave/valor, como un diccionario.
2. Listas de valores.

Se agrupan estos elementos dentro de objetos, definidos por llaves ('{', '}'), y se separan utilizando comas.

```
{
    "nombre": "valor",
    "objeto": {
        "lista": [
          1, true, "otro_valor"
        ]
    }
}
```

- Las listas, al igual que los valores, son precedidas por una clave.
- Los valores pueden ser números, cadenas de caracteres, otros objetos, listas, etc. Una explicación detallada de la notación se puede conseguir aquí.



A través de llamadas HTTP, se puede transferir información entre distintas aplicaciones utilizando JSON

UTILIDAD

1. Su composición permite agrupar todo tipo de datos que una aplicación pueda necesitar.
2. Son sencillos de utilizar. Interpretadores de JSON han sido implementados por un gran número de lenguajes de programación.

Microservicios

autentia

¿Qué es?

Un microservicio es una **aplicación pequeña que ejecuta su propio proceso y se comunica mediante mecanismos ligeros** (normalmente una API de recursos HTTP). Cada aplicación se encarga de implementar una funcionalidad completa del negocio, es desplegado de forma independiente y puede estar programado en distintos lenguajes así como usar diferentes tecnologías de almacenamiento de datos.

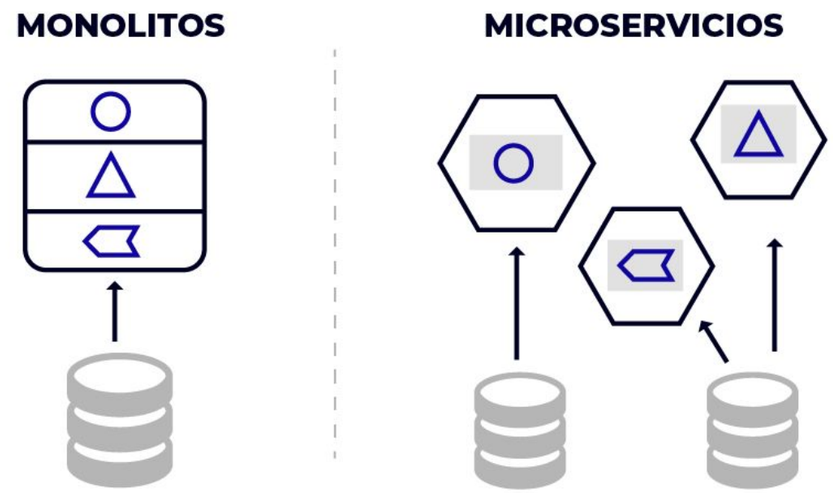
CARACTERÍSTICAS PRINCIPALES

Hasta hace unos años, las aplicaciones grandes y complejas se implementaban como grandes monolitos muy difíciles de mantener y evolucionar. Una arquitectura basada en microservicios **consiste en implementar los distintos servicios de nuestra aplicación a través de servicios más pequeños e independientes entre sí.**

Estos servicios se caracterizan por:

- Poco acoplamiento.
- Mantenibilidad.
- Totalmente independientes del resto de microservicios.
- Cada uno implementa una arista de la aplicación de negocio (p. ej. microservicio de gestión de usuarios).

La decisión de si utilizar o no este tipo de diseño a la hora de construir nuestro sistema, **se fundamenta básicamente en el nivel de complejidad que va a alcanzar.** Para una aplicación con una complejidad baja no se recomienda utilizar esta arquitectura.



VENTAJAS DE LOS MICROSERVICIOS

- Escalabilidad más eficiente e independiente.
- Pruebas más concretas y específicas.
- Posibilidad del uso de distintas tecnologías e implementaciones.
- Desarrollos independientes y paralelos.
- Aumento de la tolerancia a fallos.
- Mejora de la mantenibilidad.
- Permite el despliegue independiente.

¿Qué es?

Los microservicios son un patrón de diseño software a nivel arquitectónico que **implementan los servicios** ofrecidos por una aplicación **mediante la colaboración de otros servicios más pequeños y autónomos**.
Fuente: https://martinfowler.com/articles/microservices.html

CARACTERÍSTICAS PRINCIPALES

- **Componentes vía Servicios:** conecta componentes como si de piezas de un puzle se tratasen. Estos componentes son unidades de software que son independientemente reemplazables y actualizables. Se pretende que estos componentes se desglosen en servicios y evitar hacer llamadas a funciones internas como se hace tradicionalmente.
- **Organizada alrededor de las funcionalidades del negocio:** el enfoque consiste en dividir el sistema en servicios que estén organizados bajo una funcionalidad del negocio, volviéndose muy importante conocer los límites de responsabilidad de cada uno. Como consecuencia, los equipos son multifuncionales y disponen de las habilidades necesarias para su desarrollo.
- **Productos, no proyectos:** un equipo debe hacerse cargo de un servicio durante todo el ciclo de vida de este. En lugar de ver el software como un conjunto de funcionalidades a ser completadas, existe una relación continua donde la pregunta es: ¿cómo puede el software ayudar a sus usuarios a mejorar la funcionalidad del negocio?
- **Gobierno descentralizado:** Podemos decidir utilizar diferentes lenguajes de programación y tecnologías dentro de cada servicio. De esta forma, podemos elegir la herramienta adecuada para cada uno
- **Gestión de datos descentralizado**: cada microservicio gestiona sus propios datos evitando que el resto de servicios accedan directamente a ellos forzando su acceso mediante las operaciones que ofrezca el mismo.
- **Diseño tolerante a fallos:** las aplicaciones necesitan ser diseñadas de modo que puedan tolerar los fallos de los distintos servicios.
- **Automatización de la infraestructura:** los equipos que desarrollan con la arquitectura de microservicios emplean habitualmente el enfoque de Integración y entrega continua.
- **Diseño evolutivo:** los servicios deben estar diseñados para que se sean fácilmente actualizables y/o reemplazables sin que esta evolución afecte a los consumidores.

¿Qué es?

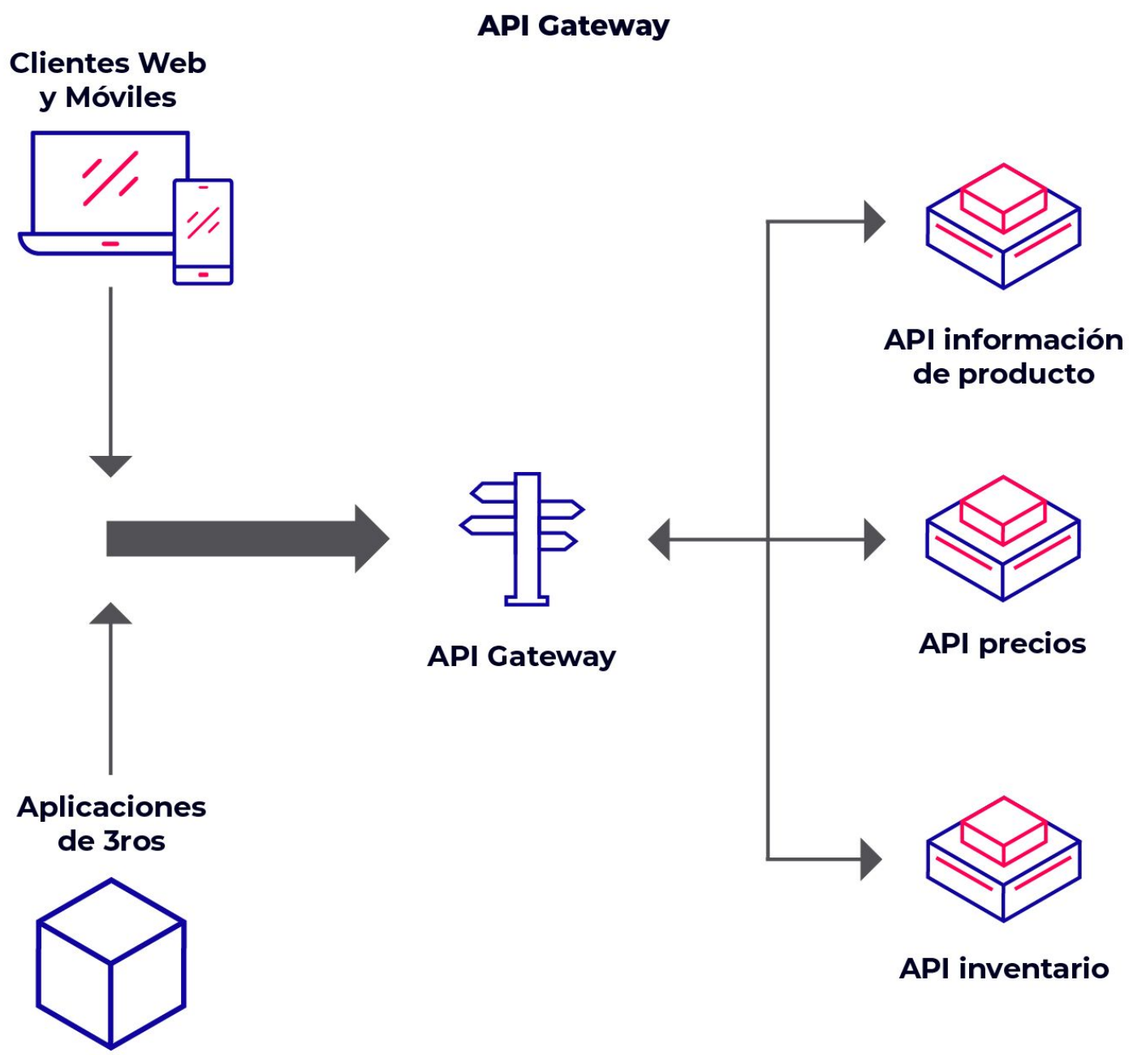
Sistema intermediario que proporciona una interfaz para hacer de enrutador entre los servicios y los consumidores desde un **único punto de entrada**. Es similar al patrón estructural Facade.

CONCEPTO

Si pensamos en una arquitectura de servicios distribuidos, habrá numerosos clientes que necesitarán intercomunicarse para completar las operaciones que se les soliciten. A medida que el número de servicios crece, es importante un intermediario que simplifique la comunicación entre los distintos clientes y servicios del sistema, en lugar de hacerlo de forma directa.

Sin ningún elemento de intermediación, cada elemento debe resolver de manera individual todas las operativas relacionadas a la comunicación. **He aquí donde entra en juego el API Gateway, delegando en este dicha complejidad, proporcionando entre otras:**.

- **Políticas de seguridad** (autenticación, autorización), protección contra amenazas (inyección de código, ataques de denegación de servicio).
- **Enrutamiento.**
- **Monitorización** del tráfico de entrada y salida.
- **Escalabilidad.**



¿Qué es?

Es un framework JVM (Java Virtual Machine) que nos permite crear nuestras **aplicaciones basadas en microservicios de una manera sencilla y rápida**. Además de Java, Micronaut permite el uso de otros lenguajes como Groovy y Kotlin.

DATOS

Los microservicios aparecieron por primera vez en 2011 y desde entonces ha aumentado su uso, intentando dejar atrás los monolitos. Esta arquitectura se basa en el despliegue de distintos servicios independientes comunicados entre sí. Micronaut es un proyecto de código abierto que ha sido desarrollado por los creadores del Framework Grails y nació para **facilitar el desarrollo de microservicios**.

Tiene una curva de aprendiza baja y sobre todo, es muy similar a Spring en cuanto a sintaxis, por lo que la gente que venga de Spring no tendrá ningún problema en adaptarse.

Las siguientes métricas se han obtenido de la documentación oficial de Micronaut a partir del desarrollo de una api rest sencilla. Se observa como Micronaut necesita más tiempo para compilar pero a cambio tiene un arranque más rápido debido a que la inyección de dependencias la hace en tiempo de compilación.

Métricas	Micronaut 2.0	Spring 2.3
Tiempo de compilación	1,48s	1,33s
Tiempo de arranque (dev)	420 ms	920 ms
Tiempo de arranque (prod)	510 ms	1,24 s

CARACTERÍSTICAS

Alguna de sus principales características son:

- **Inyección de dependencias en tiempo de compilación** en vez de en ejecución. Esto permite que el arranque de las aplicaciones sea más rápido y un bajo uso de memoria.
- Las API de Micronaut está **basada en Spring y en Grails** para ayudar a los desarrolladores a optimizar y avanzar de forma óptima en sus requerimientos.
- **Desarrollo rápido y sencillo de microservicios**, basándose en anotaciones, al igual que Spring.
- **Programación reactiva** a través de RxJava y ProjectReactor.
- Framework ideal para el **desarrollo de aplicaciones cloud nativas,** ya que soporta el uso de herramientas para el descubrimiento de servicios, como Eureka y Consul, y sistemas distribuidos como Zipkin y Jaeger.
- Soporta la **integración con GraalVM.**
- Uso mínimo de proxys.
- Pruebas unitarias sencillas.

MICRONAUT

GraalVM

autentia

¿Qué es?

Es una **JVM** y **JDK** basada en HotSpot/OpenJDK e implementada en Java. Soporta lenguajes de programación adicionales y modos de ejecución, como la compilación ***ahead-of-time*** que permite un **tiempo de arranque más rápido** en aplicaciones Java, resultando en ejecutables que **ocupan menos memoria.**

OBJETIVOS

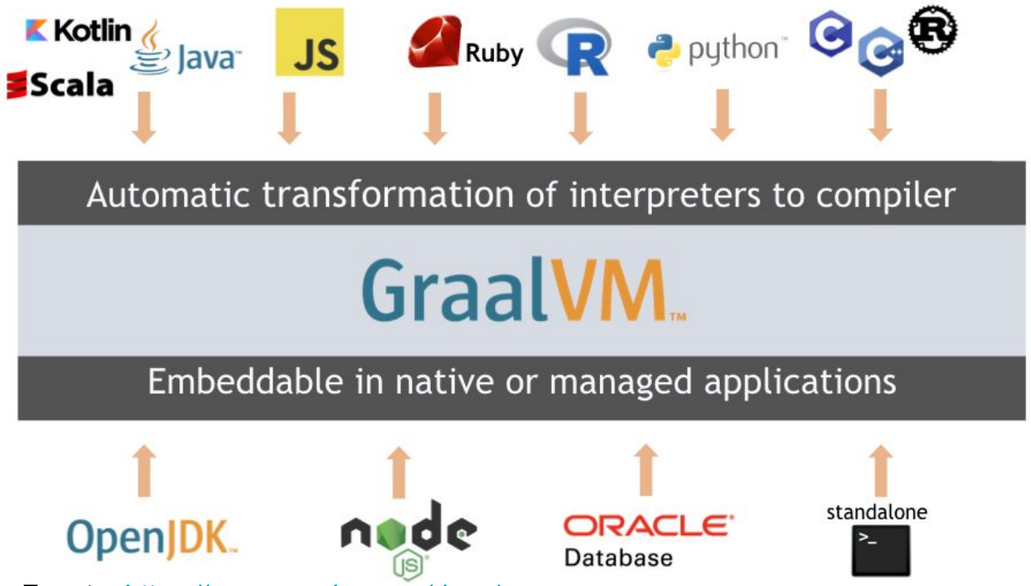
- Mejorar el **rendimiento** de los lenguajes basados en la máquina virtual de Java haciendo que tengan un rendimiento similar a los lenguajes nativos.
- **Reducir el tiempo de arranque** de las aplicaciones de la JVM mediante la compilación *ahead-of-time* (antes de tiempo) con GraalVM Native image.
- Permitir la integración de GraalVM en Oracle Database, OpenJDK, Node.js, Android/iOS y otros similares.
- Permitir utilizar código de cualquier lenguaje **en una única aplicación.**
- Incluir una colección de herramientas, fácilmente extensible, para la programación de **aplicaciones políglotas**.

COMPONENTES

- GraalVM Compiler: se trata de un compilador **JIT** para Java.
- GraalVM Native Image: permite la compilación ***ahead-of-time.***
- Truffle Language Implementation Framework: depende de GraalVM SDK y permite implementar **otros lenguajes** en GraalVM.
- Instrumentation-based Tool Support: soporte para **instrumentación dinámica**, que es agnóstica del lenguaje.

LENGUAJES Y RUNTIMES

- GraalVM JavaScript: runtime de **JavaScript** ECMAScript 2019, con soporte para Node.js.
- TruffleRuby: implementación de **Ruby** con soporte preliminar para Ruby on Rails.
- FastR: implementación del lenguaje **R**.
- GraalVM Python: implementación de **Python 3**.
- GraalVM LLVM Runtime (SuLong): implementación de un intérprete de bitcode **LLVM**.
- GraalWasm: implementación de **WebAssembly**.



Fuente: https://www.graalvm.org/docs/

¿Qué son?

Son las principales herramientas para la gestión de proyectos: compilación, tests, gestión de dependencias, integración con herramientas de integración continua, despliegue o generación de releases.

MAVEN

- Configuración del proyecto a través de ficheros XML (pom.xml).
- Gestión de dependencias clara y sencilla.
- Definición de repositorios de donde descargar las dependencias del proyecto.
- Ciclo de vida basado en fases predefinidas. Cada fase contiene goals (p. ej. Empaquetado, ejecución de tests...).
- Es la herramienta de gestión de configuración más utilizada en el desarrollo de proyectos software.
- Amplio catálogo de plugins desarrollados.
- Gran soporte por parte de la comunidad.
- Soporta la integración con herramientas de integración continua como Jenkins o Travis.

GRADLE

- Configuración del proyecto a través de ficheros escritos en Groovy o Kotlin (build.gradle).
- Configuración basada en proyectos y tareas. Un proyecto representa qué se quiere hacer y está compuesto de varias tareas.
- Como se tiende a programar tareas propias, los scripts pueden variar de un proyecto a otro.
- Gestión de dependencias.
- Soporta la integración con herramientas de integración continua como Jenkins o Travis.
- Integración con otras herramientas de gestión como Maven o Ant.

¿Qué es?

La programación reactiva es un paradigma basado en el desarrollo declarativo por contra al tradicional, que es imperativo. Se trata de funcionar de una forma **no bloqueante, asíncrona y dirigida por eventos**.

¿QUÉ PROBLEMA RESUELVE?

Se trata de evitar ciertos problemas que se han visto que pueden suceder con las arquitecturas tradicionales, que suelen funcionar de una forma bloqueante. Esto puede, en ocasiones, desaprovechar la CPU, ya que los hilos se encuentran bloqueados por entrada salida (ir a base de datos, consultar el API de un tercero...), incluso puede llegar a la saturación del sistema y finalmente su caída con volúmenes de carga altos. La programación reactiva se sustenta en la base de **NIO**. Con NIO en vez de un pool de hilos en la que cada hilo procesa una petición de inicio a fin, vamos a tener hilos workers. Estos workers van a coger las peticiones de los usuarios y en vez de esperar cuando se bloqueen, van a utilizarse para servir otras peticiones, aprovechando al máximo nuestra CPU. Es decir, vamos a tener un escalado vertical óptimo. Tanto en on premise como en cloud. Esto puede **suponer un ahorro en costes de infraestructura**. Lo recomendable es que el número de workers sea igual que el número de cores que tenga nuestra CPU.

Con las nuevas arquitecturas, como por ejemplo los microservicios, nos podemos ver en la necesidad de hacer múltiples llamadas entre ellos, lo que puede desembocar en una maraña de mensajes que tienen que ir en cierto orden para al final producir una respuesta. Si a eso le sumamos el protocolo de comunicaciones HTTP (que es síncrono) podemos tener problemas de rendimiento.

En el 2014 surge el manifiesto de los sistemas reactivos. Podemos ver conceptos como que los sistemas tienen que ser **responsivos** (responder rápido en la medida de lo posible), **resilientes** (tolerante a fallos), **elásticos** (adaptable a carga) **y orientados a mensajes** (de forma asíncrona). La programación reactiva nos ayuda solo en el último de estos pasos y en algunos casos en el primero. Uno de los conceptos claves será establecer un **mecanismo de backpressure**, que nos va a permitir limitar los mensajes por parte de los productores para que los consumidores no se saturen.

Se está viendo cada vez una adopción más de este paradigma y solo hace falta mirar lenguajes como Java, que desde la versión 9 incorpora Reactive Streams o frameworks como Spring que también lo incluyen en su módulo webflux. La programación reactiva no debe usarse para todo. Se recomienda para escenarios con tiempos de respuesta superiores al segundo cuyo tiempo se pierde casi en la totalidad en peticiones bloqueantes.

Kotlin

¿En qué consiste?

Paradigma de programación **declarativa** en la que las **funciones** son ciudadanas de primera clase.

CARACTERÍSTICAS

- **Funciones de primera clase y de orden superior**
  - Las funciones pueden recibir y devolver otras funciones.
  - Una función puede asignarse a una variable.
- **Funciones puras**
  - No tienen efectos secundarios.
  - Dado un parámetro de entrada devuelven siempre el mismo resultado.
- **Programación declarativa**
  - Expresamos qué queremos hacer y no el cómo.
- **No hay estructuras de control**
  - Se utiliza la recursividad para resolver problemas en los que se utilizarían estas estructuras tradicionalmente.
- **Inmutabilidad**
  - Los lenguajes puramente funcionales simulan el estado pasando datos inmutables entre las funciones.

LENGUAJES

Algunos lenguajes adoptan este paradigma por completo y todas las funciones son **puras**, lo que significa que no hay estado mutable como tal, ni se permiten efectos secundarios.

Por otra parte, algunos lenguajes, llamados **impuros** adoptan parcialmente la programación funcional, permitiendo el uso de características propias funcionales junto con las de otros paradigmas. En los impuros, podremos escribir parte del código en un estilo funcional.

Algunos ejemplos de lenguajes funcionales son:

Puros:

- Haskell.
- Miranda.

Impuros:

- Scala.
- Python.
- Java (a partir del 8).
- Kotlin.
- Rust.

¿Qué les diferencia?

Los lenguajes de alto nivel utilizan una sintaxis más parecida a la natural, a cómo nos comunicamos los humanos, mientras que los de bajo nivel utilizan unas instrucciones y sentencias más parecidas al lenguaje que hablan las máquinas.

COMPARATIVA

Ventajas de los lenguajes de alto nivel:

- El **código** es **muchísimo más mantenible** ya que si está bien escrito, es posible ,a veces a simple vista, detectar errores.
- Al ser mucho más utilizados, existen muchas más herramientas de apoyo al desarrollo y cuentan con una comunidad mucho más amplia. Esto repercute en **mayores facilidades y herramientas** de desarrollo (IDEs, sistemas de debug...).
- **Desarrollar** programas complejos **es muchísimo más rápido** ya que una sentencia de alto nivel puede englobar decenas de bajo nivel.

Ventajas de los lenguajes de bajo nivel:

- Normalmente **son más rápidos** que los lenguajes de alto nivel y se suelen utilizar cuando el rendimiento extremo es una necesidad primordial.
- Suelen ocupar menos espacio y tienen **menos requisitos para su ejecución**, por lo que son ideales para dispositivos con poco espacio o poca memoria.

EJEMPLOS DE BAJO NIVEL

- Lenguaje de máquina.
- Lenguaje ensamblador.

EJEMPLOS DE ALTO NIVEL

- Java.
- Python.
- Swift.
- Kotlin.
- Javascript.
- C#.
- Go.
- y muchos más menos conocidos.

¿Qué es?

Kotlin es un lenguaje de programación multiplataforma diseñado para ser completamente interoperable con Java, pero siendo mucho más conciso. En una encuesta hecha por StackOverflow a casi 100,000 desarrolladores, Kotlin resultó ser uno de los lenguajes de programación más queridos por los desarrolladores.

HISTORIA

JetBrains, comenzó el desarrollo de Kotlin en 2010. En julio de 2011 se presentó el proyecto públicamente y en febrero de 2012 todo el código de Kotlin se hizo open source.

La versión 1.0 de Kotlin se lanzó en febrero de 2015. Esta versión es considerada la primera estable y es la versión a partir de la cual habrá retrocompatibilidad a largo plazo.

En 2017 Google anunció soporte para Kotlin en Android, poniéndolo al mismo nivel de importancia que Java. En 2019 Google anunció que Kotlin es su lenguaje preferido para desarrollar en Android, por encima de Java.

En 2020 la mayoría de grandes empresas (Pinterest, Twitter, Netflix, Uber, AirBnB, Trello, etc.) han migrado a Kotlin para el desarrollo de sus aplicaciones en Android.

KOTLIN VS. JAVA

Kotlin se diseñó como un lenguaje de programación moderno de calidad industrial pensado para superar ciertas limitaciones de Java y para permitir desarrollos más rápidos y más robustos. Kotlin es completamente interoperable con Java, permitiendo una migración gradual a este lenguaje y una curva de aprendizaje sencilla.

Debido a la gran popularidad de Java, es poco probable que en un futuro cercano Kotlin reemplace a Java por lo que lo más probable es que los dos lenguajes coexistan, haciendo que el ecosistema de Java sea más completo y mejor.

VENTAJAS

- Acabar las líneas con punto y coma **";" es opcional.**
- **Concisión:** Kotlin reduce el conocido como código boilerplate, haciendo que el código sea más rápido de escribir, más legible y menos propenso a errores. Por ejemplo, escribir una clase con sus setters, getters, equals, etc. es tan sencillo como esto:

```
data class Customer(val name: String, val email: String)
```

- **Seguridad:** Kotlin te protege de los NullPointerExceptions, ya que por defecto los tipos no aceptan valores nulos y te avisan en tiempo de compilación de errores. Tienes la opción de añadir explícitamente que tu variable acepta valores nulos usando "?". Además Kotlin hace automáticamente castings cuando compruebas el tipo de una variable.

```
var output: String
output = null   // Compilation error

val name: String? = null    // Nullable type
println(name.length())      // Compilation error

if (obj is Invoice)
    obj.calculateTotal() // auto-casts obj to Invoice
```

- **Interoperabilidad:** desde Kotlin puedes llamar directamente a cualquier clase de Java y viceversa. Kotlin puede compilar a la JVM o JavaScript.
- **Flexibilidad:** existen miles de herramientas e IDEs que facilitan el desarrollo en Kotlin.

Para saber más puedes ver este tutorial, la guía oficial o la guía de Android.

¿En qué se diferencia?

El sistema de tipos de Kotlin está hecho para eliminar el peligro de encontrarse de repente con una referencia nula en el código. Salvo que se exprese explícitamente, los nulos en Kotlin no existen. Pero existen los tipos nullables y los tipos no nullables.

Los tipos nullables

En Kotlin para poder asignar a un objeto el valor null hay que declararlo como un tipo nullable utilizando el operador ?.

```
var s1: String = "Hola, mundo"
s1 = null // el error!!!
var s2: String? = null //es un tipo nullable
```

Kotlin utiliza la misma JVM que Java, por eso también lanza la excepción NullPointerException. Las posibles causas son: llamada explícita para lanzar NullPointerException, uso del operador Not Null Assertion (!!), interoperaciones con el código Java etc.

Operador de acceso seguro (?.)

Si intentamos acceder a una variable nullable tendremos un error de compilación, porque el compilador exige usar con las variables nullables llamadas seguras..

```
var s1: String? = "Hola, mundo"
println(s1.length) // error!!
getUserData()?.parseToAnotherObject()?.execute()
```

También reduce la necesidad de usar if/else para comprobar si la variable es nula y ejecuta una acción sólo cuando la referencia tiene un valor no nulo.

Operador Not Null Assertion (!!)

El operador Not Null Assertion (!!) convierte cualquier valor en un tipo no nulo y lanza una excepción *NullPointerException* si el valor es nulo.

```
var str : String?  = "Alejandro"
println(str!!.length)
str = null
str!!.length //error!!!
```

Operador Elvis (?:)

Se utiliza para devolver un valor predeterminado si la variable es nula. Operador Elvis hace que el código sea más compacto.

```
var name = firstName?: "Desconocido"
val name = firstName ?: throw
IllegalArgumentException("Introduce el nombre correcto")
```

Estas fichas son un recurso vivo,
¡anímate a hacernos sugerencias y comentarios!

@autentia

Puedes descargar la última versión en:

https://autentia.com/libros/

Además encontrarás libros gratuitos, guías y recursos útiles para tu día a día.