RESUMEN BACKEND

GITHUB

Introducción a Git

Git es un sistema de control de versiones distribuido.

Cada usuario tiene una copia completa del repositorio en su computadora local.

Permite a los desarrolladores trabajar en ramas sin afectar el trabajo de otros.

Commits en Git

Un commit es una instantánea del repositorio en un momento dado.

Almacena cambios realizados en uno o varios archivos.

Permite retroceder en el tiempo y ver el historial de cambios.

Ramas en Git

Los repositorios pueden tener múltiples ramas para diferentes líneas de desarrollo.

Los desarrolladores pueden crear ramas para nuevas características o experimentos.

Las ramas se pueden fusionar con la rama principal una vez que están listas.

Resolución de Conflictos

Git proporciona herramientas para resolver conflictos cuando dos desarrolladores cambian el mismo archivo.

Muestra diferencias entre versiones y permite elegir qué cambios conservar.

Colaboración en Git

Facilita el trabajo en equipo y la gestión de proyectos.

Permite compartir ramas y commits con otros miembros del equipo.

Integración con plataformas como GitHub para facilitar la colaboración.

Componentes de Git

Incluye el sistema de control de versiones, una interfaz de línea de comandos (Git Bash), y una GUI.

Documentación y herramientas auxiliares para facilitar la instalación y uso.

Repositorios

Un repositorio es un espacio de almacenamiento de versiones de archivos.

Se puede crear un nuevo repositorio o clonar uno existente.

Clonar descarga el historial completo de commits, ramas y etiquetas.

Stages (Add & Reset)

El área de preparación es donde se preparan los archivos para ser confirmados.

Comandos: **git add** para añadir archivos y **git reset** para quitar archivos del área de preparación.

Commits

Para realizar un commit, se deben agregar cambios al área de preparación y luego confirmar.

Ejemplo de comandos:

git add archivo1.txt archivo2.txt

git commit -m "Mensaje del commit"

Push

git push se utiliza para subir cambios locales a un repositorio remoto.

Se requiere permisos de escritura en el repositorio remoto.

Status

git status muestra el estado actual de los archivos en el repositorio.

Indica archivos modificados, en el área de preparación y no rastreados.

Fusiones (Merge)

Permite combinar dos ramas en un único conjunto de cambios.

Comando para fusionar: **git merge**.

Fork

Un fork es una copia de un repositorio en una cuenta de usuario diferente.

Permite realizar cambios sin afectar el repositorio original.

Pull Request (PR)

Solicitud de colaboración para proponer cambios en un repositorio remoto.

Implica crear una rama, realizar cambios, y solicitar revisión y aprobación.

MAVEN

Introducción a Maven

Herramienta de gestión de proyectos para construir y gestionar proyectos Java.

Ventajas: Maneja dependencias automáticamente, simplifica la construcción y despliegue de aplicaciones.

Conceptos Básicos de Maven

Proyecto: Unidad de trabajo que representa una colección de archivos.

POM (Project Object Model): Archivo XML que contiene información del proyecto (nombre, versión, dependencias).

Dependencias: Bibliotecas necesarias para el proyecto.

Repositorios: Lugares donde se almacenan dependencias (locales y remotos).

Ciclo de Vida: Secuencia de fases para construir y desplegar software (clean, validate, compile, test, package, install, deploy).

Plugins

Definición: Extienden la funcionalidad de Maven.

Ejemplo de Plugins:

- maven-compiler-plugin: Compila el código fuente.
- maven-surefire-plugin: Ejecuta pruebas unitarias.

Arquetipos

Plantillas para crear nuevos proyectos con estructura predefinida.

Ejemplo de Arquetipos:

- maven-archetype-quickstart: Para proyectos Java simples.
- maven-archetype-webapp: Para proyectos web.

LENGUAJE JAVA

1. Introducción al Lenguaje Java

Lenguaje de programación de alto nivel, orientado a objetos, diseñado para ser portable y seguro.

Características:

Lenguaje de Alto Nivel: Sintaxis cercana al lenguaje humano, abstrae detalles del hardware.

Portabilidad: Programas pueden ejecutarse en diferentes sistemas operativos sin recompilación gracias a la JVM (Java Virtual Machine).

Orientación a Objetos: Organiza el código en objetos que representan entidades del mundo real.

2. Java Development Kit (JDK)

Componentes Principales:

Compilador (javac): Convierte código fuente en bytecode.

JVM: Interpreta el bytecode y lo ejecuta en cualquier plataforma.

Bibliotecas de Clases: Proporcionan funciones predefinidas para facilitar el desarrollo.

4. Sintaxis de Java

Reglas de Sintaxis:

Archivos .java para código fuente, .class para bytecode.

Nombres de variables y métodos deben comenzar con letra y no pueden ser palabras reservadas.

Java distingue entre mayúsculas y minúsculas.

5. Tipos de Datos

Primitivos: int, boolean, char, float, etc.

Referencia: Tipos de datos que son objetos.

- String: Representa una secuencia de caracteres.
- Arrays: Colecciones de elementos del mismo tipo.

Variables en Java

Son ubicaciones de memoria que almacenan datos. Cada variable tiene un tipo de dato, un identificador (nombre) y un valor.

Tipos de Variables

Primitivas: Tipos de datos básicos que no son objetos.

- byte: 8 bits, rango de -128 a 127.
- short: 16 bits, rango de -32,768 a 32,767.
- int: 32 bits, rango de -2,147,483,648 a 2,147,483,647.
- long: 64 bits, rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.
- float: 32 bits, para números decimales.

- double: 64 bits, para números decimales de mayor precisión.
- char: 16 bits, para un solo carácter Unicode.
- boolean: Representa valores verdadero/falso (true/false).

Ámbito de las Variables

1) Ámbito de Clase: Variables que pertenecen a la clase y son accesibles desde cualquier método de la clase.
2) Ámbito de Método: Variables locales a un método, solo accesibles dentro de ese método.
3) Ámbito de Parámetro: Variables que se pasan a un método como argumentos.
4) Ámbito de Bloque: Variables declaradas dentro de un bloque de código (como un bucle o condicional).
5) Ámbito Estático: Variables que pertenecen a la clase y son compartidas entre todas las instancias.

Operadores en Java

Símbolos que realizan operaciones sobre uno o más operandos para producir un resultado.

Tipos de Operadores

1) Aritméticos: Realizan operaciones matemáticas.
    - Suma (+): a + b
    - Resta (-): a – b
    - Multiplicación (*): a * b
    - División (/): a / b (división entera si ambos son enteros).
    - Módulo (%): a % b (resto de la división).

2) Asignación: Asignan valores a variables.
    - Asignación simple (=): a = 5
    - Asignación compuesta (+=, -=, *=, /=): a += 3 (equivale a a = a + 3).

3) Comparación: Comparan dos valores y devuelven un booleano.
    - Igual que (==): a == b
    - Distinto de (!=): a != b
    - Mayor que (>): a > b
    - Menor que (<): a < b

4) Lógicos: Combinan expresiones booleanas.
    - AND lógico (&&): a && b
    - OR lógico (||): a || b
    - Negación lógica (!): !a

5) Incremento y Decremento: Aumentan o disminuyen el valor de una variable en 1.
    - Incremento (++): a++ o ++a
    - Decremento (--): a-- o –a

6) Ternario: Operador condicional que evalúa una expresión booleana.
    - Sintaxis: condición ? valorSiVerdadero : valorSiFalso

COLECCIONES

Introducción

Estructuras de datos para almacenar y manipular conjuntos de elementos.

Arreglos

- Unidimensionales: Almacenan elementos en una fila.
- Multidimensionales: Almacenan elementos en filas y columnas.

Colecciones

- Listas: Estructuras dinámicas.
- ArrayList: Basada en un arreglo.
- LinkedList: Basada en nodos enlazados.

Mapas

Colección de pares clave-valor.

- HashMap: No ordenado.
- TreeMap: Ordenado.
- LinkedHashMap: Mantiene el orden de inserción.

Conjuntos

Colección de elementos únicos.

- HashSet: No ordenado.
- TreeSet: Ordenado.
- LinkedHashSet: Mantiene el orden de inserción.

TESTING

Introducción

Proceso de evaluar el comportamiento de un sistema o módulo de software para asegurar que funciona adecuadamente y cumple con los requerimientos especificados.

Herramientas: JUnit y Mockito son las más utilizadas para realizar pruebas en Java.

Fundamentos del Testing

Objetivo: Identificar errores y mejorar la calidad del software.

Tipos de Testing

1) Pruebas Unitarias: Verifican el funcionamiento de unidades de código (clases, métodos).
2) Pruebas de Integración: Verifican la interacción entre diferentes unidades de código.
3) Pruebas de Sistema: Verifican que el software cumple con los requerimientos en el entorno de producción.
4) Pruebas de Aceptación: Realizadas por el cliente para verificar que el software satisface sus necesidades.
5) Pruebas de Regresión: Verifican que cambios en el software no afecten funcionalidades previamente probadas.

3. Tests Unitarios

Comprobación del correcto funcionamiento de unidades de código de forma aislada.

- Ejemplo en JUnit:

```
@Test
void testAdd() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.add(2, 3));
}
```

4. Mockito

Biblioteca para crear objetos simulados (mocks) en pruebas unitarias.

Uso: Permite simular el comportamiento de objetos o dependencias externas.

4.1 @Mock

Anotación que se utiliza para crear un objeto simulado de una clase o interfaz.

- Ejemplo:

```
public class MyClassTest {
    @Mock
    private MyDependency myDependency;
    ...
}
```

4.2 @Spy

Crea un objeto simulado a partir de un objeto real, manteniendo su comportamiento original.

- Ejemplo:

```java
public class MyClassTest {
  @Spy
  private MyClass mySpy;

  // ...
}
```

4.3 @InjectMocks

Inyecta automáticamente los mocks y spies en una instancia de la clase que se está probando.

- Ejemplo:

```java
public class MyClassTest {

    @Mock
    private MyDependency myDependency;

    @InjectMocks
    private MyClass myClass;

    @Test
    public void testMyMethod() {
        // Configurar el comportamiento del mock
        when(myDependency.doSomething()).thenReturn("Resultado esperado");

        // Ejecutar el método que estamos probando
        String result = myClass.myMethod();

        // Verificar el resultado
        assertEquals("Resultado esperado", result);
    }
}
```

5. Reflections

Técnica que permite a un programa inspeccionar y manipular objetos en tiempo de ejecución, incluso si no se conoce su tipo en tiempo de compilación.

Uso en Testing: Permite acceder a métodos y campos privados para realizar pruebas unitarias.

6. Pruebas en Métodos Privados

Los métodos privados no se pueden probar directamente, pero se pueden usar técnicas como la reflexión para acceder a ellos.

- Ejemplo:

```java
import org.junit.platform.commons.support.ReflectionSupport;

MiClase instancia = new MiClase();

ReflectionSupport.invokeMethod(
    ReflectionSupport.findMethod(MiClase.class, "metodoPrivado"),
    instancia);
```

ALGORITMOS

¿Qué es un algoritmo?

Un conjunto de pasos ordenados y finitos que permiten resolver un problema o realizar una tarea específica.

Propiedades

1) Claro y preciso: Cada paso debe estar bien definido.
2) Debe tener un fin: No puede ejecutarse indefinidamente.
3) Efectivo: Cada paso debe ser realizable en un tiempo razonable y con los recursos disponibles.

Clasificación de Algoritmos

1) Por tipo de problema:

- Algoritmos de búsqueda: Encuentran un elemento en una estructura de datos.
    - Búsqueda lineal: O(n) - Recorre cada elemento.
    - Búsqueda binaria: O(log n) - Requiere que la lista esté ordenada.

- Algoritmos de ordenamiento: Reorganizan elementos.
    - Bubble sort: $O(n^2)$ - Simple pero ineficiente.
    - Merge sort: O(n log n) - Divide y conquista.
    - Quick sort: O(n log n) - Muy usado en la práctica.

- Algoritmos de optimización: Buscan la mejor solución posible.

- Algoritmos de recorrido: Exploran estructuras de datos.

2) Por forma de operar:

- Algoritmos iterativos: Usan bucles (for, while).

- Algoritmos recursivos: Se llaman a sí mismos.

- Por eficiencia esperada:
    - Notación Big O: Describe el crecimiento del tiempo de ejecución.

Complejidad de los Algoritmos

Cantidad de recursos computacionales necesarios para resolver un problema.

Recursos a analizar

- Tiempo: Cuánto tarda el algoritmo en ejecutarse.
- Espacio: Cuánta memoria necesita durante su ejecución.

Ejemplo de complejidades

- Mejor caso: O(1) (ejemplo: encontrar un elemento al inicio).
- Peor caso: O(n) (ejemplo: recorrer toda la lista).
- Caso promedio: O(n/2) (ejemplo: búsqueda lineal).

Tipos de Complejidad

1) Complejidad Temporal: Mide el tiempo de ejecución en función del tamaño de entrada (n).

2) Complejidad: O(n).

3) Complejidad: $O(n^2)$.

Notación Big O

Expresa la tasa de crecimiento del tiempo de ejecución.

- O(1): Tiempo constante.
- O(log n): Tiempo logarítmico (ejemplo: búsqueda binaria).
- O(n): Tiempo lineal.
- O(n log n): Tiempo lineal logarítmico (común en algoritmos de ordenamiento).
- $O(n^2)$: Tiempo cuadrático (ejemplo: comparación de todos contra todos).
- $O(2^n)$, O(n!): Tiempo exponencial o factorial (impracticable para entradas grandes).

1. Introducción a Spring Boot

Qué es y por qué usarlo

Spring Boot es una herramienta que extiende el framework Spring para facilitar y agilizar la creación de aplicaciones Java independientes y listas para producción. Su objetivo es reducir al mínimo el esfuerzo de configuración inicial. Las tres funcionalidades principales que lo hacen tan efectivo son:

- **Configuración Automática:** Spring Boot inicializa las aplicaciones con dependencias preestablecidas, configurando automáticamente tanto el framework Spring subyacente como las librerías de terceros necesarias, basándose en las mejores prácticas y reduciendo la posibilidad de errores.
- **Enfoque Obstinado (Opinionated):** El framework toma decisiones de configuración por defecto sobre qué paquetes instalar y qué valores usar. Esto se logra mediante los "Spring Starters", que son dependencias preconfiguradas para casos de uso típicos (ej. "spring-web" para aplicaciones web), permitiendo al desarrollador enfocarse en la lógica de negocio.
- **Aplicaciones Independientes (Standalone):** Permite empaquetar una aplicación como un archivo JAR ejecutable que integra un servidor web (como Tomcat). Esto elimina la necesidad de desplegar en un servidor de aplicaciones externo y permite que la aplicación se ejecute "simplemente" con un comando.

Ciclo de Vida del Proyecto

El flujo de trabajo para un proyecto con Spring Boot se resume en los siguientes pasos:

1. **Creación:** El proyecto se genera utilizando herramientas como Spring Initializr (vía web) o Spring Boot CLI (línea de comandos), donde se seleccionan las dependencias (starters) necesarias.

2. **Configuración de Propiedades:** Las configuraciones específicas de la aplicación, como la conexión a la base de datos o el puerto del servidor, se definen en archivos externos como application.properties o application.yml, sin modificar el código fuente.
3. **Ejecución:** La aplicación puede ser ejecutada desde un IDE, mediante la línea de comandos (spring run) o como un JAR ejecutable (java -jar).
4. **Despliegue:** El método más común es desplegar el archivo JAR ejecutable. Alternativamente, se puede generar un archivo WAR para ser desplegado en un contenedor de servlets tradicional.

---

2. Desarrollo de Aplicaciones con Spring Boot

Desarrollo de APIs

Spring Boot simplifica enormemente la creación de APIs RESTful mediante el módulo Spring MVC y un modelo de programación basado en anotaciones. Se utilizan controladores (@RestController) para definir los endpoints que procesan las peticiones HTTP. La documentación de estas APIs se facilita con herramientas como **Swagger**, que, a través de librerías de integración como Springfox, genera automáticamente una especificación **OpenAPI** y una interfaz de usuario interactiva a partir de las anotaciones del código.

Acceso a Datos

El documento describe dos enfoques principales para la manipulación de datos:

- **Spring Data JPA:** Proporciona una capa de alta abstracción sobre el framework JPA. Mediante la definición de interfaces de repositorio que extienden JpaRepository, Spring Data implementa automáticamente las operaciones CRUD (Crear, Leer, Actualizar, Borrar), eliminando la necesidad de escribir consultas SQL manualmente.
- **Spring JDBC:** Ofrece un enfoque de más bajo nivel, simplificando el uso del JDBC estándar de Java. Se utilizan clases como JdbcTemplate para ejecutar consultas SQL escritas manualmente, lo que proporciona mayor control y flexibilidad a costa de más código explícito.

Integración y Pruebas

- **Integración:** La comunicación con servicios externos (ej. consumo de APIs RESTful) se realiza utilizando bibliotecas como RestTemplate o el más moderno WebClient. Spring Boot también soporta la integración con bases de datos externas, servicios SOAP y proveedores de autenticación.
- **Pruebas:** Spring Boot se integra nativamente con JUnit. La estrategia de pruebas se basa en:
    - **Pruebas Unitarias:** Se enfocan en componentes aislados, utilizando mocks (objetos simulados) con la anotación @MockBean para simular dependencias externas.

    - **Pruebas de Integración:** Se utilizan anotaciones como @SpringBootTest para cargar el contexto completo de la aplicación y probar la interacción entre capas, incluyendo llamadas HTTP a los controladores (@WebMvcTest) y la persistencia de datos (@DataJpaTest).

---

3. Resumen de Anotaciones Clave

Las anotaciones son fundamentales en Spring Boot. Se pueden agrupar por su funcionalidad:

- **Anotaciones Core y de Configuración:** Definen la estructura de la aplicación y gestionan los componentes.
    - @SpringBootApplication: Anotación principal que habilita la autoconfiguración y el escaneo de componentes.
    - @Configuration, @Component, @Service, @Repository: Estereotipos que marcan clases para ser gestionadas por Spring como beans. Se usan para clases de configuración, componentes genéricos, lógica de negocio y acceso a datos, respectivamente.
    - @Bean: Declara explícitamente un bean dentro de una clase de configuración.
- **Anotaciones Web (Controladores y REST):** Gestionan las peticiones HTTP.
    - @RestController: Marca una clase como un controlador REST, donde los métodos devuelven datos directamente en el cuerpo de la respuesta.
    - @RequestMapping (y variantes @GetMapping, etc.): Mapean las URLs y métodos HTTP a los métodos del controlador.
    - @PathVariable, @RequestBody: Extraen datos de la URL y del cuerpo de la petición, respectivamente.
- **Anotaciones de Persistencia (JPA):** Definen el mapeo objeto-relacional (ORM).
    - @Entity, @Table: Marcan una clase como una entidad persistente y especifican su tabla en la base de datos.
    - @Id: Identifica el campo que actúa como clave primaria.
    - @OneToMany, @ManyToOne, etc.: Definen las relaciones (uno a muchos, muchos a uno) entre las entidades.
- **Anotaciones Funcionales/Utilitarias:** Añaden comportamientos transversales.
    - @Transactional: Asegura que un método se ejecute dentro de una transacción de base de datos.
    - @Async: Permite que un método se ejecute en un hilo separado (de forma asíncrona).
    - @Scheduled: Configura un método para que se ejecute periódicamente.

---

4. Arquitectura de Microservicios

Fundamentos

- **Definición:** Un enfoque arquitectónico donde una aplicación se compone de servicios pequeños, independientes y débilmente acoplados. Cada servicio tiene su propia tecnología, se organiza por capacidad de negocio y se comunica mediante APIs, eventos o brokers de mensajes.
- **Beneficios y Desafíos:**
    - **Beneficios: Despliegue independiente** de cada servicio, **escalado preciso** solo de los componentes que lo necesitan y la libertad de usar "la herramienta adecuada para el trabajo" en cada servicio.
    - **Desafíos:** Mayor complejidad de gestión, problemas de latencia y conectividad, y la necesidad de una monitorización robusta en un sistema distribuido.
- **Cultura DevOps:** Es un **requisito** fundamental para el éxito. La complejidad de gestionar múltiples servicios hace imprescindible la automatización del ciclo de vida (CI/CD), la supervisión y la implementación.

Ecosistema Tecnológico

El documento destaca las siguientes herramientas clave para una arquitectura de microservicios:

- **Contenedores (Docker, Kubernetes):** Docker empaqueta los servicios en contenedores ligeros. Kubernetes se ha convertido en la solución estándar para la orquestación, gestión y escalado de estos contenedores.
- **API Gateways:** Actúan como una capa intermedia o punto de entrada único que enruta las peticiones de los clientes a los servicios correspondientes, proporcionando además seguridad y autenticación.
- **Mensajería (Messaging):** Para la comunicación asíncrona, se acoplan llamadas API con sistemas de mensajería (como brokers de mensajes de uso general) o streaming de eventos (como Apache Kafka) para que los servicios puedan emitir y escuchar cambios de estado.

Patrones y Antipatrones

- **Patrones de Diseño Comunes:**
    1. **Backend for Frontend (BFF):** Crea un backend específico por cada tipo de interfaz de usuario (ej. uno para móvil, otro para web) para optimizar la experiencia en lugar de usar un backend genérico.
    2. **Patrón Strangler (Estrangulador):** Ayuda a refactorizar una aplicación monolítica migrando gradualmente su funcionalidad a nuevos microservicios, como una vid que "estrangula" lentamente al árbol original.
    3. **Descubrimiento de Servicios (Service Discovery):** Proporciona mecanismos para que las instancias de servicio, que cambian dinámicamente, puedan encontrarse entre sí.
- **Antipatrones (Malas Prácticas a Evitar):**
    1. **Crear microservicios demasiado pronto:** No empezar con microservicios. Son una solución a la complejidad de un monolito grande; si esa complejidad no existe, introducen problemas innecesarios.

2. **Hacerlos demasiado pequeños ("Nanoservicios"):** Exagerar el "micro" puede llevar a una sobrecarga y complejidad que supera los beneficios.
3. **No tener DevOps:** Intentar gestionar microservicios sin una automatización adecuada de la implementación y supervisión es buscar problemas.