Ce este spring IOC?

Spring IOC (Inversion of Control) este un principiu de proiectare și o caracteristică cheie a cadrului de lucru Spring.

În esență, Spring IOC presupune transferul controlului asupra fluxului de execuție a aplicației de la aceasta însăși către un container

extern (Spring container). Scopul principal al IOC este de a obține un cuplaj slab între diferitele componente ale aplicației, făcând-o

astfel mai modulară, ușor de întreținut și mai ușor de testat.

În programarea tradițională, aplicația creează și gestionează direct obiectele de care depinde. Cu IOC, responsabilitatea de a crea și

gestiona obiectele este delegată unui container extern. Containerul este conștient de configurarea și dependențele obiectelor și le

injectează în obiecte în timpul rulării aplicației. Acest proces este cunoscut și sub denumirea de injecție de dependențe (DI).

Ce este spring ioc container?

Containerul Spring IOC este componenta de bază a cadrului Spring responsabilă de gestionarea obiectelor, a creării și a dependențelor

acestora. Containerul citește metadatele de configurare (de obicei în XML sau prin adnotări) și creează și gestionează obiectele

(numite și "boabe" sau "beans") conform configurației.

Ce este BeanFactory?

BeanFactory: Acesta este versiunea mai simplă a containerului și oferă suport de bază pentru DI și gestionarea ciclului de viață al bean-urilor.

Ce este SpringApplicationContext?

ApplicationContext: Acesta este un container mai complex și oferă funcționalități avansate, construite deasupra BeanFactory.

Include facilități pentru internaționalizare (i18n), propagarea evenimentelor, integrarea cu Spring AOP (Aspect-Oriented Programming) și altele.

ce este @ComponentScan?

@ComponentScan este o adnotare din cadrul Spring Framework care este utilizată pentru a specifica pachetele din aplicație care

trebuie scanate pentru a găsi clasele componente și a le înregistra în Spring IoC Container. Această adnotare face parte din suita de

adnotări de configurare a cadrului Spring și este folosită împreună cu @Configuration.

Atunci când utilizați @ComponentScan, Spring va căuta toate clasele din pachetele specificate / daca nu este specificat un arguemnt se ia pachetul

prezent si toate subpachetele și va identifica clasele componente pe care le-ați marcat cu adnotări precum @Component, @Service, @Repository,

@Controller, etc. Acestea sunt denumite și clase marcate ca fiind "stereotipuri" și sunt considerate componente în Spring.

Ce este @AutoConfiguration?

@EnableAutoConfiguration, este o altă adnotare din cadrul Spring Framework, iar rolul său este de a activa și de a utiliza funcționalități de c

onfigurare automată (auto-configuration) pentru aplicația Spring.

Atunci când adăugați adnotarea @EnableAutoConfiguration în clasa principală a aplicației Spring, Spring Boot va scana

claselor din classpath și va încerca să configureze implicit bean-urile și alte elemente necesare pentru funcționalitățile

specifice detectate în aplicație. Acest lucru înseamnă că nu mai trebuie să configurați manual multe dintre componentele

comune sau să definiți fișiere XML de configurare.

Ce este @Configuration?

@Configuration este o adnotare din cadrul Spring Framework și are rolul de a marca o clasă ca fiind o clasă de configurare a Spring.

În alte cuvinte, când adăugați adnotarea @Configuration la o clasă, Spring va recunoaște acea clasă ca fiind o sursă de configurare

pentru contextul aplicației.

Clasele marcate cu @Configuration pot conține metode anotate cu @Bean. Aceste metode sunt folosite pentru a defini și a configura

bean-urile care vor fi gestionate de Spring IoC Container. Când containerul inițializează contextul aplicației, va detecta clasele

marcate cu @Configuration și va procesa metodele marcate cu @Bean, creând și înregistrând bean-urile corespunzătoare în container.

Diferenta dintre @Bean si @Component?

În concluzie, @Component este o adnotare generică folosită pentru a marca clasele ca fiind componente, în timp ce @Bean este o adnotare

specifică Spring utilizată pentru a defini metode de fabrică pentru bean-uri în clasele marcate cu @Configuration. Ambele adnotări contribuie

la configurarea contextului aplicației și a bean-urilor gestionate de Spring IoC Container.

ce este o clasa immutabila? de ce string este immutabil? cum se creeaza o clasa immutabila?

O clasă imutabilă este o clasă a cărei stare (starea internă) nu poate fi modificată după ce obiectul a fost creat. În alte cuvinte,

odată ce un obiect imutabil a fost creat, valorile sale nu pot fi schimbate prin metodele publice ale clasei. Orice operație care

ar modifica starea obiectului va crea de fapt un nou obiect, păstrând astfel integritatea obiectului original.

Acum, să discutăm despre de ce tipul String din limbajul Java este imutabil. Stringurile sunt imutabile în Java pentru a aduce mai

multe beneficii și pentru a evita comportamente nedorite. Cateva dintre motivele pentru care Stringurile sunt imutabile sunt:

Securitate: Datorită imutabilității, stringurile pot fi utilizate în mod sigur ca chei în colecții, cum ar fi hărți (Maps) și seturi

(Sets), fără a afecta integritatea acestora.

Potențialul de a fi folosite în hash-uri: Stringurile sunt frecvent utilizate ca chei în tabelele hash, cum ar fi HashMap. Imutabilitatea

asigură că valoarea hash a unui string nu se schimbă în timp, ceea ce este important pentru a asigura o recuperare corectă a datelor din

colecții bazate pe hash.

Utilizarea în sincronizare: Imutabilitatea permite utilizarea stringurilor în mod sigur în medii concurente și în operații de sincronizare

fără a introduce probleme de acces concurent la date.

Eficiență: Stringurile imutabile pot fi stocate în cache pentru reutilizare, deoarece acestea nu se schimbă.

Pentru a crea o clasă imutabilă în Java, urmați următoarele principii:

Declarați clasa ca final, astfel încât să nu poată fi extinsă și modificată de alte clase.

Declarați toate câmpurile ca fiind private și final, astfel încât valorile lor să nu poată fi schimbate după crearea obiectului.

Nu oferiți metode publice care să modifice starea internă a obiectului (adică să nu aveți metode de tip "setter").

Asigurați-vă că orice modificare a stării obiectului creează un nou obiect și nu modifică starea obiectului original.

Dacă obiectul conține referințe la alte obiecte mutabile, asigurați-vă că aceste referințe nu pot fi schimbate sau că sunt gestionate

în mod corespunzător pentru a menține imutabilitatea generală a obiectului.

ce este java reflection?

Java Reflection este o caracteristică a limbajului Java care permite unui program să inspecteze și să manipuleze dinamic clase, obiecte,

metode și câmpuri în timpul execuției. Cu alte cuvinte, Java Reflection oferă posibilitatea de a "reflecta" (analiza) structura internă a

claselor și obiectelor la runtime, fără a cunoaște această structură la momentul compilării.

cum lucreaza intern un hashmap? avantaje si dezavantaje

HashMap este o implementare a interfeței Map din Java, utilizată pentru a stoca și gestiona perechi cheie-valoare. Intern, un HashMap utilizează o structură de date numită tablou de dispersie (hash table) pentru a eficientiza căutările și operațiile de adăugare și ștergere a elementelor.

Cum funcționează intern un HashMap:

În primul rând, fiecare cheie este transformată într-un cod hash utilizând metoda hashCode() a obiectului cheie.

Acest cod hash este utilizat pentru a determina indexul în tabloul de dispersie unde va fi stocată perechea cheie-valoare.

După calcularea codului hash, HashMap folosește un algoritm de dispersie (hashing) pentru a determina un index în tabloul de dispersie.

Dacă două chei au același cod hash, ele vor fi plasate în aceeași poziție din tabloul de dispersie, într-o structură d

e date numită "bucket" (găleată). Bucket-ul poate fi văzut ca o listă sau un arbore binar (în funcție de implementarea concretă).

Când căutăm o valoare asociată unei chei, HashMap va utiliza același algoritm pentru a găsi indexul în tablou și apoi va

ăuta în bucket-ul corespunzător pentru a găsi valoarea.

Dacă dimensiunea tabloului de dispersie devine insuficientă în raport cu numărul de elemente, HashMap va redimensiona tabloul

intern și va reorganiza elementele pentru a evita coliziunile și a menține o performanță optimă.

Avantaje ale HashMap:

Performanță constantă pentru operațiile de bază: În medie, operațiile de adăugare, căutare și ștergere într-un HashMap au o complexitate de timp constantă O(1). Aceasta înseamnă că timpul de execuție al acestor operații este constant și nu crește odată cu creșterea dimensiunii map-ului.

Structură flexibilă: HashMap poate stoca orice tip de obiecte ca valori și poate fi utilizat pentru o varietate de scenarii.

Dezavantaje ale HashMap:

Ordinea nedefinită: Un HashMap nu garantează ordinea în care perechile cheie-valoare vor fi iterare. În schimb, puteți folosi LinkedHashMap

dacă doriți să mențineți ordinea de inserție.

Coliziuni de hash: Dacă două chei au același cod hash, se va produce o coliziune. Pentru a gestiona coliziunile, HashMap trebuie să utilizeze

bucket-uri suplimentare, ceea ce poate duce la o scădere a performanței în cazul în care au loc coliziuni frecvente.

Consum de memorie: HashMap-ul are nevoie de o cantitate suplimentară de memorie pentru a stoca structurile de date interne (tabloul de dispersie

și bucket-urile). În cazul în care dimensiunea map-ului este foarte mare, consumul de memorie poate deveni un dezavantaj.

cum lucreaza intern un arraylist? avantaje si dezavantaje

ArrayList este o clasă din cadrul colecțiilor Java care implementează interfața List și este utilizată pentru a stoca o

colecție de elemente într-un mod ordonat. Intern, ArrayList utilizează un tablou dinamic pentru a stoca elementele și permite adăugarea,

căutarea și ștergerea acestora într-un mod eficient.

Cum funcționează intern un ArrayList:

La început, ArrayList creează un tablou intern de dimensiune zero sau cu o dimensiune inițială specificată.

Când se adaugă un element în ArrayList, acesta este inserat la sfârșitul tabloului intern. Dacă tabloul intern este plin și nu mai

are suficient spațiu pentru a adăuga un nou element, ArrayList va crește dimensiunea tabloului intern pentru a face loc elementului nou.

Dacă se șterge un element din ArrayList, aceasta va reorganiza tabloul intern pentru a umple golul lăsat de elementul șters.

ArrayList este o clasă din cadrul colecțiilor Java care implementează interfața List și este utilizată pentru a stoca o colecție de elemente într-un mod ordonat. Intern, ArrayList utilizează un tablou dinamic pentru a stoca elementele și permite adăugarea, căutarea și ștergerea acestora într-un mod eficient.

Cum funcționează intern un ArrayList:

La început, ArrayList creează un tablou intern de dimensiune zero sau cu o dimensiune inițială specificată.

Când se adaugă un element în ArrayList, acesta este inserat la sfârșitul tabloului intern. Dacă tabloul intern este plin și nu mai are suficient spațiu pentru a adăuga un nou element, ArrayList va crește dimensiunea tabloului intern pentru a face loc elementului nou.

Dacă se șterge un element din ArrayList, aceasta va reorganiza tabloul intern pentru a umple golul lăsat de elementul șters.

Avantaje ale ArrayList:

Accesul rapid la elemente: Deoarece ArrayList se bazează pe un tablou intern, accesul la elementele sale este foarte rapid, având o complexitate de timp constantă O(1). Accesul la un element la o anumită poziție se realizează într-un mod direct, deoarece tabloul intern folosește indexarea.

Eficiența în cadrul operațiilor specifice: ArrayList este eficient pentru operațiile de adăugare și ștergere a elementelor la sfârșitul listei (complexitate O(1)). Operațiile de adăugare și ștergere în interiorul listei au complexitate O(n), unde n este numărul total de elemente din listă. Cu toate acestea, operațiile de adăugare sau ștergere la sfârșitul listei sunt mai frecvente, deci acestea sunt optimizate.

Dezavantaje ale ArrayList:

Performanță scăzută pentru operații de adăugare și ștergere în interiorul listei: Dacă trebuie să adăugați sau să ștergeți frecvent

elemente în interiorul listei (nu la sfârșitul ei), operațiile de reorganizare a tabloului intern pot fi costisitoare, deoarece necesită

mutarea tuturor elementelor din stânga sau din dreapta poziției respective.

Complexitate în creșterea dimensiunii: Când dimensiunea tabloului intern trebuie crescută pentru a face loc unui nou element, ArrayList

trebuie să aloce un nou tablou, să copieze elementele din tabloul vechi în cel nou și să elibereze memoria ocupată de tabloul vechi.

Acest proces poate fi costisitor, mai ales atunci când lista este foarte mare.

Consum de memorie: Dacă ArrayList are o dimensiune inițială mare și are puține elemente, există riscul ca lista să ocupe mai multă memorie

decât este necesară pentru a stoca elementele efective.

cum lucreaza intern un linkedlist? avantaje si dezavantaje

LinkedList este o altă implementare a interfeței List din Java, dar diferă de ArrayList prin modul în care stochează și gestionează elementele.

În loc să utilizeze un tablou intern, LinkedList utilizează o structură de date bazată pe liste dublu înlănțuite (doubly linked list), unde fiecare element este conectat la elementele anterioare și următoare din listă.

Cum funcționează intern un LinkedList:

Fiecare element (nod) din LinkedList conține o referință la elementul următor și la elementul anterior din listă.

În mod similar cu ArrayList, adăugarea unui element la finalul listei este rapidă, deoarece doar nodul curent și ultimul nod trebuie să fie

conectate pentru a crea o nouă legătură între ele.

Adăugarea sau ștergerea unui element în interiorul listei implică actualizarea referințelor pentru nodurile anterioare și următoare pentru a conecta noul nod sau a elimina nodul existent. Deoarece LinkedList are referințe către nodul anterior și următor, acest lucru poate fi realizat mai rapid decât în ArrayList.

Avantaje ale LinkedList:

Adăugarea și ștergerea rapidă a elementelor în interiorul listei: Datorită structurii sale dublu înlănțuite, LinkedList este mai eficient în

adăugarea și ștergerea elementelor în interiorul listei decât ArrayList, deoarece nu necesită realocarea tabloului intern.

Performanță mai bună pentru operațiile de adăugare și ștergere frecvente: Dacă aplicația necesită adăugarea și ștergerea frecventă a elementelor

în interiorul listei, LinkedList poate fi o alegere mai bună decât ArrayList.

Dezavantaje ale LinkedList:

Accesul lent la elemente: Datorită faptului că nu se bazează pe indexare directă ca ArrayList, accesul la elementele din LinkedList poate fi mai

lent și are complexitate O(n), unde n este numărul de elemente din listă. Pentru a accesa un anumit element, va trebui să parcurgeți lista

începând de la început sau de la sfârșit, în funcție de poziția dorită.

Consum de memorie: Fiecare element din LinkedList conține referințe către elementul următor și anterior, ceea ce duce la un consum mai mare de

memorie decât în ArrayList, unde doar un tablou intern este folosit.

Indicarea la obiecte Garbage Collector-ului: Datorită faptului că fiecare nod din LinkedList conține referințe, aceasta poate duce la mai multe

indicii pentru colectorul de gunoi (Garbage Collector), ceea ce poate afecta performanța în aplicațiile cu multe operații de adăugare și ștergere.

ce este set and hashset? avantaje si dezavantaje

În Java, Set și HashSet sunt interfețe și, respectiv, o implementare specifică a interfeței Set, care reprezintă o colecție de elemente distincte, fără duplicări. O caracteristică cheie a seturilor este că acestea nu permit stocarea duplicatelor, astfel încât fiecare element să fie unic în set.

Set:

Set este o interfață din cadrul colecțiilor Java care extinde interfața Collection.

Seturile nu permit elemente duplicate și nu au o ordine specifică de păstrare a elementelor.

Principalele metode ale interfeței Set includ add(), remove(), contains(), isEmpty(), size(), etc.

HashSet:

HashSet este o clasă din cadrul colecțiilor Java care implementează interfața Set.

HashSet utilizează o structură de date bazată pe tablou de dispersie (hash table) pentru a stoca și a gestiona elementele.

Elementele în HashSet nu au o ordine specifică și pot fi stocate în orice ordine, în funcție de funcția hash a elementelor.

Avantaje ale HashSet:

Elemente unice: HashSet asigură că fiecare element este unic în set și nu permite duplicarea acestora.

Adăugare rapidă: Adăugarea unui element în HashSet este rapidă, deoarece se bazează pe mecanismul de dispersie (hashing), ceea ce face

ca operația de inserție să aibă o complexitate medie de timp constantă O(1).

Căutare rapidă: Căutarea unui element în HashSet are, de asemenea, o complexitate medie de timp constantă O(1), datorită utilizării tabloului

de dispersie.

Dezavantaje ale HashSet:

Ordinea nedefinită: Elementele din HashSet nu sunt păstrate într-o ordine specifică, ceea ce înseamnă că nu aveți control asupra modului

în care elementele vor fi iterați sau afișate.

Utilizare de memorie: HashSet utilizează o cantitate suplimentară de memorie pentru a stoca tabloul de dispersie și pentru a evita coliziunile.

Itinerarea lentă: Din cauza lipsei unei ordini definite, itinerarea prin elementele HashSet poate fi mai lentă decât itinerarea prin alte tipuri

de colecții, cum ar fi LinkedHashSet sau TreeSet, unde elementele sunt menținute într-o anumită ordine.

cum compari 2 obiecte?

Compararea a două obiecte implică determinarea dacă acestea sunt egale sau nu. În limbajul de programare Java, comparația a două obiecte

poate fi realizată în două moduri diferite, în funcție de ceea ce înseamnă "egalitate" în contextul aplicației:

Compararea referințelor: Acest tip de comparație se realizează prin verificarea dacă cele două obiecte au aceeași referință în memorie. Adică,

verificăm dacă cele două variabile referă același obiect. În Java, operatorul == este folosit pentru comparația referințelor:

Compararea conținutului: Acest tip de comparație se realizează prin verificarea dacă valorile conținute de cele două obiecte sunt egale.

Pentru a realiza această comparație, trebuie să supraîncărcați metoda equals() din clasa obiectului. Aceasta permite implementarea proprie

de comparație între obiecte. În multe clase din biblioteca standard Java, metoda equals() a fost deja supraîncărcată pentru a face o comparație

adecvată, cum ar fi în cazul claselor String, Integer, etc.:

ce scope-uri cunosti in spring boot?

În Spring Boot, există mai multe scope-uri (domenii de vizibilitate) pe care le puteți utiliza pentru a defini modul în care Spring IoC Container

gestionează și creează instanțele componentelor și a altor bean-uri în cadrul aplicației. Fiecare scope determină cât de des se creează noi

instanțe ale unui bean și cât de mult timp vor fi menținute în container. Iată cele mai frecvent utilizate scope-uri în Spring Boot:

Singleton (implicit): Acesta este scope-ul implicit în Spring și asigură că o singură instanță a bean-ului este creată și utilizată în întreaga

aplicație. Este cel mai frecvent utilizat pentru majoritatea componentelor și serviciilor, deoarece oferă o eficiență mai bună și reduce overhead-ul.

Prototype: Un bean cu scope Prototype va crea o nouă instanță de fiecare dată când este solicitat de către aplicație sau de alte bean-uri. Acesta

oferă o instanțiere non-Singleton, utilă pentru situațiile în care doriți să aveți o instanță nouă și independentă la fiecare injectare.

Request: Un bean cu scope Request este creat o dată pentru fiecare cerere HTTP și este disponibil pe toată durata vieții cererii.

Acest scope este potrivit pentru utilizarea în aplicații web pentru a gestiona date specifice unei cereri.

Session: Un bean cu scope Session este creat o dată pentru fiecare sesiune a utilizatorului și rămâne în memoria sesiunii până când sesiunea

expiră sau utilizatorul se deconectează. Este util pentru a gestiona date specifice sesiunii utilizatorului în aplicații web.

Application: Un bean cu scope Application este creat o singură dată și rămâne în memoria aplicației pe toată durata vieții acesteia. Acesta

este potrivit pentru a gestiona date globale în cadrul aplicației.

Pentru a seta un anumit scope, puteți utiliza anotarea @Scope și specifica numele scope-ului dorit ca argument.

ce este singleton ?

În contextul Spring Boot, Singleton se referă la modul în care Spring gestionează și creează obiecte de tipul componentelor și serviciilor

implicite (de exemplu, clasele annotate cu @Component, @Service, etc.) în cadrul aplicației.

În Spring Boot, majoritatea componentelor și serviciilor sunt gestionate ca Singleton-uri, ceea ce înseamnă că o singură instanță a acestora

este creată și folosită în întreaga aplicație. Acest lucru se datorează modului în care Spring Boot administrează și gestionează componentele și

serviciile prin intermediul Spring IoC Container (Inversion of Control Container).

Atunci când creați o clasă și o marcați cu una dintre anotările componente din Spring (@Component, @Service, etc.), aceasta devine parte a

contextului de aplicație administrat de Spring IoC Container. În momentul în care aplicația rulează și necesită instanțe ale componentelor

respective, Spring IoC Container le va crea o singură dată și va menține aceste instanțe pe toată durata vieții aplicației.

ce este builder pattern? cum il implementezi?

Builder Pattern este un pattern de proiectare (design pattern) de creare care este folosit pentru a construi și configura obiecte complexe

cu mulți parametri, astfel încât să se evite un constructor cu un număr mare de argumente sau combinații de constructori (constructor telescopic).

Prin intermediul acestui pattern, se separă procesul de construire a obiectului de reprezentarea internă a acestuia, permițând o construcție mai

flexibilă și ușurând citirea și întreținerea codului.

Cum funcționează Builder Pattern:

public class Produs {

private String parametru1;

private int parametru2;

// Alte atribute și metode ale clasei

}

public interface Builder {

Builder setParametru1(String value);

Builder setParametru2(int value);

Produs build();

}

public class ConcreteBuilder implements Builder {

private String parametru1;

private int parametru2;

@Override

public Builder setParametru1(String value) {

this.parametru1 = value;

return this;

}

@Override

public Builder setParametru2(int value) {

this.parametru2 = value;

return this;

}

@Override

public Produs build() {

return new Produs(parametru1, parametru2);

}

}

Obiect Complex: Acesta este obiectul rezultat al construcției, care este de obicei returnat de către metoda build() din ConcreteBuilder.

ce este o tranzactie?

În contextul bazelor de date și al sistemelor informatice, o tranzacție reprezintă o unitate logică și atomică de lucru, care constă

într-o serie de operații (citire, scriere, actualizare, ștergere) efectuate asupra bazelor de date sau altor resurse, pentru a aduce sistemul

dintr-un stadiu coerent în altul.

Caracteristicile principale ale unei tranzacții sunt:

Atomicitate: O tranzacție este atomică, ceea ce înseamnă că toate operațiile care fac parte din tranzacție sunt tratate ca o singură

unitate indivizibilă. Dacă una dintre operații eșuează, toate operațiile efectuate până în acel moment sunt anulate (rollback), iar baza de

date rămâne în starea inițială (consistență).

Consistență: O tranzacție aduce baza de date dintr-un stadiu valid în altul. Acest lucru înseamnă că, după finalizarea tranzacției, datele

din baza de date vor respecta regulile și constrângerile de integritate definite.

Izolare: O tranzacție funcționează izolat față de alte tranzacții. Modificările făcute în cadrul unei tranzacții nu sunt vizibile altor

tranzacții până când tranzacția curentă nu este finalizată.

Durabilitate: Odată ce o tranzacție este finalizată (comisă), modificările efectuate asupra bazei de date devin permanente și trebuie să

fie persistente chiar și în cazul întreruperilor de sistem sau a eșecurilor hardware.

ce sunt genericile?

În Java, genericile reprezintă o caracteristică a limbajului care permite definirea și utilizarea de clase, interfețe și metode

care pot lucra cu tipuri de date generice, cunoscute sub numele de tipuri generice. Prin intermediul genericelor, puteți crea clase

sau metode care să poată manipula și să funcționeze cu mai multe tipuri de date, fără a specifica tipul de date specific la momentul compilării.

Beneficiile utilizării genericelor în Java includ:

Reutilizare și flexibilitate: Prin definirea claselor și metodelor cu genericitate, puteți crea cod care să poată fi reutilizat și

adaptat pentru diverse tipuri de date, fără a rescrie același cod pentru fiecare tip în parte.

Siguranța tipurilor (Type Safety): Utilizarea genericelor asigură că tipurile de date sunt verificate la momentul compilării, prevenind

astfel erorile de tip. Acest lucru permite detectarea erorilor mai devreme în procesul de dezvoltare, înainte ca aplicația să fie executată.

Eliminarea necesității de conversie manuală: Genericele permit manipularea directă a datelor de tip specific, fără a fi nevoie să convertiți

tipurile de date în mod manual.

Cod mai clar și mai ușor de înțeles: Prin folosirea genericelor, codul devine mai clar și mai ușor de înțeles, deoarece este mai abstractizat

și nu se bazează pe tipuri de date specifice.

ce este record?

Un record este în mod implicit imutabil, adică câmpurile sale nu pot fi modificate după ce obiectul a fost creat. De asemenea,

record-urile generează automat implementările standard pentru metodele equals(), hashCode() și toString(), pe baza valorilor câmpurilor.

Aceste implementări se bazează pe valoarea de egalitate a câmpurilor și nu pe referința obiectelor.

record Person(String name, int age) {

// Nu este nevoie de cod suplimentar, toate metodele precum equals(), hashCode(), toString() sunt generate automat.

}

Record-urile sunt proiectate pentru a reduce boilerplate-ul (cod redundant) asociat cu crearea și gestionarea claselor de tip POJO

(Plain Old Java Object) folosite doar pentru a reține date. O clasă record este definită utilizând cuvântul cheie record, urmat de

numele clasei și lista de câmpuri (variabile) ale clasei.

ce sunt optionalele in java?

Caracteristici ale Optional-ului:

Este o încapsulare a valorii opționale: Optional reprezintă o încapsulare a valorii, care poate fi prezentă (există) sau absentă (nulă).

Evită NullPointerException: Prin folosirea Optional-urilor, programatorul poate evita verificarea manuală a valorii nule și posibilele

erori asociate.

Metode utile: Clasa Optional oferă o serie de metode utile pentru a manipula valorile opționale, cum ar fi isPresent(), orElse(),

orElseGet(), orElseThrow(), etc.

ce sunt exceptiile diferenta intre checked si unchecked?

Excepții verificate (Checked Exceptions):

Excepțiile verificate sunt acele excepții care trebuie să fie gestionate explicit de către programator, prin intermediul

declarației de throws în semnătura metodelor sau prin blocuri try-catch.

Acestea extind clasa Exception (sau o subclasă a acesteia), dar nu extind clasa RuntimeException.

Programatorul este obligat să gestioneze excepțiile verificate prin tratarea lor în codul său sau prin propagarea lor în sus

către apelantul metodei.

Exemple de excepții verificate includ IOException, SQLException, FileNotFoundException, etc.

Excepții neverificate (Unchecked Exceptions):

Excepțiile neverificate sunt acele excepții care nu trebuie să fie gestionate explicit în cod prin intermediul throws sau blocuri try-catch.

Acestea extind clasa RuntimeException sau o subclasă a acesteia.

Programatorul are opțiunea de a trata sau nu excepțiile neverificate, dar nu este obligat să o facă.

Dacă o excepție neverificată nu este tratată, va provoca o excepție de timpul execuției și va termina execuția programului.

Exemple de excepții neverificate includ NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, etc.

ce este docker si cum functioneaza?

Docker este o platformă open-source care utilizează tehnologia de virtualizare la nivel de sistem de operare pentru a facilita crearea,

implementarea și gestionarea aplicațiilor în containere software. Containerele Docker oferă un mod izolat și portabil pentru a rula aplicații,

împreună cu toate dependențele lor, fără a afecta sistemul gazdă.

Funcționarea Docker constă în următoarele componente principale:

Docker Engine: Acesta este motorul Docker, care rulează pe sistemul gazdă și gestionează toate operațiunile legate de containere.

Docker Engine include un server de demon care primește cereri API pentru a crea, gestiona și executa containere, precum și un CLI (Command Line Interface) pentru a comunica cu daemon-ul.

Imagini Docker: O imagine Docker este un șablon static care conține toate dependențele și instrucțiunile necesare pentru a crea un container

rulabil. Imaginile Docker sunt construite din fișiere Dockerfile, care conțin o listă de instrucțiuni pentru a instala pachete, copia fișiere,

configura mediul de lucru și așa mai departe. Imaginile Docker sunt stocate în registre Docker (precum Docker Hub) și pot fi partajate și reutilizate.

Containere Docker: Un container Docker reprezintă o instanță în execuție a unei imagini Docker. Atunci când creați un container dintr-o imagine,

acesta devine izolat de restul sistemului gazdă și conține toate dependențele necesare pentru a rula aplicația. Containerele sunt portabile,

deoarece pachetele și mediul de lucru sunt izolate și independente de sistemul gazdă.

Registre Docker: Registrele Docker sunt depozite pentru imagini Docker. Docker Hub este un registru public, unde puteți găsi imagini oficiale și

comunitare. De asemenea, puteți crea registre private pentru a stoca imagini personalizate.

Cum funcționează Docker:

Construirea Imaginilor: Pentru a crea o imagine Docker, trebuie să creați un fișier Dockerfile care conține instrucțiunile necesare pentru

configurarea mediului și instalarea dependențelor aplicației. Apoi, puteți utiliza CLI-ul Docker pentru a construi imaginea din Dockerfile.

Gestionarea Containerele: Cu CLI-ul Docker, puteți crea, porni, opri și șterge containere. Când creați un container dintr-o imagine, acesta

devine o instanță a imaginii și poate fi pornit și oprit la nevoie.

Izolarea și Portabilitatea: Containerele Docker rulează izolat de restul sistemului gazdă și au propriul mediu de lucru. Aceasta asigură

portabilitatea, deoarece pot fi implementate pe orice sistem care rulează Docker, indiferent de sistemul gazdă.

ce este kubernetes si cum functioneaza ?

Kubernetes (cunoscut și sub denumirea de K8s) este o platformă open-source de orchestrare și gestionare a containerelor. A fost dezvoltată de

Google și ulterior donată către Cloud Native Computing Foundation (CNCF). Scopul principal al Kubernetes este de a facilita implementarea,

scalarea și gestionarea aplicațiilor containerizate într-un mod eficient și automatizat.

Cum funcționează Kubernetes:

Noduri (Nodes): Clusterele Kubernetes constau dintr-o colecție de mașini fizice sau virtuale numite noduri. Fiecare nod este o mașină pe care

rulează serviciile Kubernetes, inclusiv serviciul principal numit "kubelet" care se ocupă de gestionarea containerelor pe nod.

Poduri (Pods): Celula de bază a Kubernetes este numită "pod". Un pod reprezintă o grupare logică de unul sau mai multe containere care rulează

împreună pe același nod. Aceste containere împărtășesc aceleași resurse și pot comunica între ele utilizând localhost.

Replicare și Scalare: Kubernetes facilitează replicarea și scalarea aplicațiilor în funcție de cerințele de trafic și performanță. Un controler

de replicare se asigură că numărul specificat de poduri identice este întotdeauna disponibil, iar dacă cerințele cresc sau scad, Kubernetes

poate scala automat numărul de replici pentru a face față cerințelor.

Service-uri: Kubernetes folosește servicii pentru a expune aplicațiile către rețea internă sau internet. Serviciile sunt seturi de poduri care

pot fi accesate printr-un singur IP și un nume DNS, indiferent de numărul de replici ale podului.

Volume-uri (Volumes): Kubernetes permite atașarea unor volume persistente la containere pentru a stoca datele în afara ciclului de viață al

containerelor. Astfel, datele pot fi păstrate chiar și după distrugerea și recrearea podurilor.

Declarative Configurations: Kubernetes funcționează pe baza unei abordări declarative. Utilizatorii descriu starea dorită a sistemului într-un

fișier de configurare (cum ar fi un fișier YAML) și apoi Kubernetes se asigură că sistemul ajunge în starea specificată.

Controlul automat: Kubernetes se asigură că starea actuală a sistemului corespunde cu starea declarată în fișierele de configurare.

Dacă există deviații între starea declarată și starea reală, Kubernetes va efectua acțiuni automate pentru a readuce sistemul la starea dorită.

cum poti testa in spring boot si cum mockui un obiect impreuna cu dependintele sale ?

Testele Unitare: Testele unitare verifică funcționalitatea individuală a unităților de cod, cum ar fi metodele dintr-o clasă.

Acestea sunt scrise pentru a verifica logica specifică a clasei, izolând-o de restul aplicației.

@RunWith(MockitoJUnitRunner.class)

public class UserServiceUnitTest {

@Mock

private UserRepository userRepository;

private UserService userService;

@Before

public void setUp() {

userService = new UserService(userRepository);

}

@Test

public void testGetUserById() {

// Definim comportamentul mock-ului userRepository

User user = new User(1, "John Doe");

Mockito.when(userRepository.getUserById(1)).thenReturn(user);

// Apelăm metoda getUserById din userService

User result = userService.getUserById(1);

// Verificăm rezultatul

assertEquals(user, result);

}

}

Testele de Integrare: Testele de integrare verifică modul în care componente individuale ale aplicației interacționează între ele.

Aceste teste pot implica configurări ale contextului Spring și verificarea comunicării dintre servicii.

@RunWith(SpringRunner.class)

@SpringBootTest

public class UserServiceIntegrationTest {

@Autowired

private UserRepository userRepository;

private UserService userService;

@Before

public void setUp() {

userService = new UserService(userRepository);

}

@Test

public void testGetUserById() {

// Salvăm un utilizator în baza de date pentru test

User user = new User(1, "John Doe");

userRepository.save(user);

// Apelăm metoda getUserById din userService

User result = userService.getUserById(1);

// Verificăm rezultatul

assertNotNull(result);

assertEquals(user.getId(), result.getId());

assertEquals(user.getName(), result.getName());

}

}

Testele Mock: Testele mock presupun crearea unor obiecte falsificate (mock) care imită comportamentul obiectelor reale. Acestea sunt

utilizate pentru a izola unitățile testate și pentru a se concentra doar pe logica lor specifică.

// Interfața UserRepository

public interface UserRepository {

User getUserById(int id);

}

// Clasa UserService

@Service

public class UserService {

private UserRepository userRepository;

@Autowired

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public User getUserById(int id) {

return userRepository.getUserById(id);

}

}

@RunWith(MockitoJUnitRunner.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

public void testGetUserById() {

// Definim comportamentul mock-ului userRepository

User user = new User(1, "John Doe");

Mockito.when(userRepository.getUserById(1)).thenReturn(user);

// Apelăm metoda getUserById din userService

User result = userService.getUserById(1);

// Verificăm rezultatul

assertEquals(user, result);

}

}