**Aký je rozdiel medzi Spring a Spring Boot?**

**Spring Framework**

* **Spring** je komplexný framework pre vývoj aplikácií v Jave. Poskytuje komplexnú infraštruktúru pre vývoj aplikácií, vrátane inverzie kontroly (IoC), dependency injection, aspektovo orientovaného programovania (AOP), prístupu k dátam, transakčného manažmentu a viac.
* Je navrhnutý tak, aby podporoval vývoj aplikácií s vysokou modularitou a jednoduchou testovateľnosťou.
* Vyžaduje od vývojárov, aby konfigurovali veľa aspektov svojich aplikácií, čo môže byť časovo náročné a vyžadovať hlboké pochopenie celého frameworku.

**Spring Boot**

* **Spring Boot** je rozšírenie Spring frameworku, ktoré zjednodušuje proces vývoja Spring aplikácií. Jeho cieľom je minimalizovať konfiguračnú prácu potrebnú na zbehnutie Spring aplikácie a ponúkať "out-of-the-box" konfiguráciu pre mnohé bežné vývojárske úlohy.
* Spring Boot automaticky konfiguruje Spring aplikácie na základe pridaných závislostí do projektu, čo umožňuje rýchly vývoj a nasadenie.
* Obsahuje zabudovanú podporu pre vytváranie samostatne spustiteľných aplikácií. Tieto aplikácie môžu byť spustené priamo z príkazovej riadky bez potreby deployovať na webový server.
* Ponúka rozsiahlu podporu pre mikroslužby, vrátane jednoduchej integrácie s cloudovými službami, čo uľahčuje vývoj distribuovaných systémov.

**Rozdiel**   
  
Kľúčový rozdiel medzi Spring a Spring Boot je v tom, že Spring Boot poskytuje predkonfigurované nastavenia a konvencie, ktoré umožňujú rýchlejší a jednoduchší vývoj aplikácií s menším množstvom manuálnej konfigurácie. Spring Boot je v podstate "opinionated" verzia Spring frameworku, ktorá znižuje potrebu vývojárskej konfigurácie a umožňuje rýchlejšie prototypovanie a vývoj.  
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
**Anotácia Autoconfiguration**  
  
Anotácia **@EnableAutoConfiguration** v Spring Boot je kľúčovou súčasťou, ktorá umožňuje automatickú konfiguráciu Spring aplikácie na základe pridaných závislostí v projektovom súbore **pom.xml** alebo **build.gradle**. Táto anotácia hľadá a aplikuje konfiguračné triedy (auto-configuration classes), ktoré sú preddefinované v Spring Boot, čím značne zjednodušuje proces konfigurácie aplikácie.

**Ako @EnableAutoConfiguration funguje**

* **Automatická konfigurácia**: Spring Boot sa pokúsi automaticky nakonfigurovať vašu Spring aplikáciu na základe jar súborov pridaných do vašeho classpathu. Napríklad, ak máte **spring-boot-starter-web** pridaný do vašej aplikácie, Spring Boot automaticky pridá Tomcat a Spring MVC do vašej aplikácie.
* **Podmienená konfigurácia**: Auto-konfigurácie využívajú Spring Framework anotácie ako **@ConditionalOnClass**, **@ConditionalOnBean**, **@ConditionalOnProperty** atď., aby zabezpečili, že konfigurácia sa aplikuje len vtedy, keď sú splnené určité podmienky. To znamená, že konfigurácia sa aktivuje len ak sú splnené všetky potrebné podmienky pre danú funkcionalitu.

**Príklad použitia**

Predstavme si, že vytvárate jednoduchú web aplikáciu s použitím Spring Boot. Váša **main** trieda, ktorá spúšťa aplikáciu, by mohla vyzerať takto:  
  
A computer screen shot of a program code

Description automatically generated

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.EnableAutoConfiguration;**

**import org.springframework.context.annotation.ComponentScan;**

**import org.springframework.context.annotation.Configuration;**

**@Configuration // Označuje triedu ako zdroj definícií beanov pre aplikáciu**

**@EnableAutoConfiguration // Povoluje automatickú konfiguráciu**

**@ComponentScan // Hľadá komponenty, konfigurácie a služby v balíčku, kde sa nachádza trieda, označuje ich pre skenovanie**

**public class MyApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(MyApplication.class, args); // Spustenie Spring Boot aplikácie**

**}**

**}**  
  
V tomto príklade:

* **@Configuration** označuje triedu ako zdroj bean definícií pre kontext aplikácie.
* **@EnableAutoConfiguration** povie Spring Boot, aby pridal auto-konfiguráciu na základe závislostí prítomných v projekte a na základe obsahu vašeho classpathu.
* **@ComponentScan** umožňuje automatické detekovanie a registráciu komponentov v balíčku, kde sa trieda nachádza, a v jej podbalíčkoch.

Použitím **@EnableAutoConfiguration**, Spring Boot automaticky nakonfiguruje vašu aplikáciu s predvolenými nastaveniami vhodnými pre mnohé bežné aplikácie. Môžete tiež explicitne vylúčiť niektoré auto-konfigurácie, ak potrebujete viac kontroly, pridaním **exclude** atribútu do **@EnableAutoConfiguration** anotácie.

Tento prístup umožňuje rýchly vývoj a nasadenie Spring aplikácií s minimalizovanou potrebou manuálnej konfigurácie.

**Ako by si hľadal, či máš triedu var na class...?**   
  
Otázka zo zozvučenia sa môže zdať trochu nejasná, ale zdá sa, že sa pýta na to, ako by ste zistili, či v rámci určitej triedy existuje premenná (alebo vlastnosť) s názvom **var**. Táto otázka môže testovať vaše porozumenie reflexii v Jave alebo vaše schopnosti práce s metadátami tried v rámci jazyka.

V Jave sa na zisťovanie informácií o triedach a ich členoch (vrátane premenných) používa reflexia. Reflexia umožňuje programu skúmať alebo "reflektovať" na svojej vlastnej štruktúre, vrátane tried, polí, metód a konštruktorov.

Tu je príklad, ako by ste mohli použiť reflexiu na zistenie, či trieda obsahuje premennú s názvom **var**:

A computer screen shot of a program code

Description automatically generated

V tomto príklade:

Používame Class<?> clazz = MyClass.class; na získanie Class objektu, ktorý reprezentuje triedu MyClass.

Potom clazz.getField("var"); sa pokúšame získať verejnú premennú (field) s názvom var. Ak taká premenná existuje, program vypíše, že premenná existuje. Inak NoSuchFieldException je vyvolaná, čo znamená, že taká premenná v triede neexistuje.

Keďže otázka na pohovore môže byť mierne nejasná, môže byť tiež dobré opýtať sa na ďalšie vysvetlenie alebo upresnenie, či sa otázka týka hľadania premennej s názvom var, hľadania všetkých premenných v triede alebo či má otázka iný zámer. Otázka tiež môže skúšať, či viete, ako dynamicky pracovať s členmi triedy v Jave, čo je užitočná schopnosť v mnohých pokročilých a reflexívnymi scenároch.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako by podľa Teba mal vyzerať dobrý kód (keď robíš code review)?**

Pri vykonávaní code review sa zameriavam na viacero kľúčových aspektov, ktoré definujú, čo považujem za "dobrý kód". Tieto aspekty zahŕňajú čitateľnosť, udržateľnosť, efektivitu, bezpečnosť a testovateľnosť. Tu je niekoľko konkrétnych bodov, ktoré hodnotím:

**1. Čitateľnosť a zrozumiteľnosť**

* **Konzistentný štýl kódovania:** Používanie jednotného štýlu písania kódu v celom projekte, vrátane závorek, odsadení, pomenovania premenných a metód.
* **Jasné pomenovania:** Premenné, funkcie, triedy a iné identifikátory by mali mať zmysluplné a opisné názvy.
* **Komentáre a dokumentácia:** Dôležité časti kódu by mali byť komentované, vysvetľujúce zložité algoritmy, použitie zložitých funkcií alebo dôvody pre určité rozhodnutia. Zároveň by dokumentácia nemala byť nadbytočná.

**2. Udržateľnosť a rozširiteľnosť**

* **Modulárny dizajn:** Kód by mal byť organizovaný do logických blokov alebo modulov, ktoré sú ľahko rozširiteľné a udržateľné.
* **Dodržiavanie princípov SOLID:** Tieto princípy pomáhajú vytvárať softvér, ktorý je ľahšie udržateľný, rozširiteľný a komponovateľný.
* **Využívanie návrhových vzorov:** Ak je to vhodné, používanie osvedčených návrhových vzorov na riešenie bežných problémov.

**3. Efektivita a výkon**

* **Optimalizácia algoritmov:** Kód by mal byť napísaný efektívne, bez zbytočných výpočtov alebo zložitosti, ktoré by mohli negatívne ovplyvniť výkon.
* **Správne využívanie zdrojov:** Dbať na správne využívanie systémových zdrojov, ako sú pamäť a procesor.

**4. Bezpečnosť**

* **Predchádzanie bežným bezpečnostným hrozbám:** Kód by mal byť napísaný s ohľadom na bezpečnosť, aby sa predišlo problémom ako sú SQL injekcie, cross-site scripting (XSS), nezabezpečené ukladanie hesiel a iné.
* **Validácia vstupov:** Vstupy od užívateľa by mali byť vždy validované a sanitizované.

**5. Testovateľnosť**

* **Testovacie prípady:** Kód by mal byť doplnený o jednotkové testy pokrývajúce kritické funkcionality a edge cases.
* **Oddelenie logiky a UI:** Uľahčuje písanie testov tým, že logika aplikácie je oddelená od užívateľského rozhrania.

**6. Dokumentácia**

* **Aktualizovaná a jasná dokumentácia:** Dokumentácia k projektu by mala byť aktuálna a poskytovať jasné usmernenia pre nových vývojárov alebo pre tých, ktorí s projektom nie sú oboznámení.

Pri vykonávaní code review hľadám kód, ktorý spĺňa tieto kritériá, zatiaľ čo zároveň rozpoznávam, že žiadny kód nie je dokonalý a že každý projekt má svoje špecifické požiadavky a obmedzenia. Cieľom je nájsť rovnováhu medzi best practices a praktickými potrebami projektu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**(Bonusova) Čo sú SOLID princípy**  
  
SOLID je akronym, ktorý označuje päť základných princípov objektovo orientovaného dizajnu a programovania. Tieto princípy pomáhajú softvérovým vývojárom vytvárať systémy, ktoré sú ľahko udržateľné, rozširiteľné a pochopiteľné. Tu sú tieto princípy detailne opísané:

**1. Single Responsibility Principle (SRP) - Princíp jedinej zodpovednosti**

* **Definícia**: Každá trieda by mala mať len jeden dôvod na zmenu. To znamená, že trieda by mala mať len jednu úlohu alebo zodpovednosť.
* **Praktický príklad**: Máme triedu **UserManager**, ktorá obsahuje metódy pre správu používateľov a zároveň zaobchádza s ukladaním používateľov do databázy. Podľa SRP by sme mali rozdeliť tieto zodpovednosti do dvoch tried: **UserManager** pre správu používateľov a **UserRepository** pre interakciu s databázou.

**-** pred SRP aplikaciou:

A computer screen shot of a black background

Description automatically generated

- po SPR aplikacii:

A computer screen with white text and red text

Description automatically generated

**2. Open/Closed Principle (OCP) - Princíp otvorenosti/zatvorenosti**

* **Definícia**: Softvérové entity (triedy, moduly, funkcie atď.) by mali byť otvorené pre rozšírenie, ale zatvorené pre zmeny. To znamená, že by malo byť možné zmeniť správanie entity bez zmeny jej zdrojového kódu.
* **Praktický príklad**: Máme systém na generovanie reportov, kde **ReportGenerator** môže generovať reporty len v jednom formáte. Ak chceme pridať nové formáty, musíme zmeniť kód **ReportGenerator**. Aby sme dodržali OCP, môžeme definovať rozhranie **ReportGenerator** a potom vytvoriť konkrétne implementácie pre každý formát reportu (napr. **PdfReportGenerator**, **HtmlReportGenerator**).

**-** pred OCP aplikáciou:

A screen shot of a computer code

Description automatically generated

- po OCP aplikácii:

A screen shot of a computer program

Description automatically generated

**3. Liskov Substitution Principle (LSP) - Liskov substitučný princíp**

* **Definícia**: Objekty v programe by mali byť nahraditeľné ich podtypmi bez ovplyvnenia správnosti programu. To znamená, že odvodené triedy musia byť schopné nahradiť svoje nadtriedy.
* **Praktický príklad**: Máme triedu **Rectangle** s vlastnosťami **width** a **height**. Trieda **Square** je odvodená z **Rectangle**, ale keďže pre štvorec platí, že **width** a **height** sú vždy rovnaké, nastavenie jednej z týchto vlastností by malo zmeniť aj druhú. Toto porušuje LSP, pretože **Square** nie je správne nahraditeľný za **Rectangle**. Riešením je redefinovať vzťah medzi **Rectangle** a **Square** tak, aby nevznikali nekonzistentnosti.

- pred LSP aplikáciou:

A screen shot of a computer program

Description automatically generated

- po LSP aplikácii:

A screen shot of a computer program

Description automatically generated

**4. Interface Segregation Principle (ISP) - Princíp segregácie rozhrania**

* **Definícia**: Klienti by nemali byť nútení závisieť na rozhraniach, ktoré nepoužívajú. To znamená, že je lepšie mať viacero špecifických rozhraní, než jedno všeobecné.
* **Praktický príklad**: Máme rozhranie **Worker** s metódami **work** a **eat**. Nie všetky pracujúce objekty však môžu potrebovať metódu **eat** (napríklad roboti). Podľa ISP by sme mali rozhranie rozdeliť na **Workable** a **Eatable**, čím umožníme objektom implementovať len tie rozhrania, ktoré potrebujú.

- pred ISP aplikáciou:

A screenshot of a computer program

Description automatically generated

- po ISP aplikácii:  
A screenshot of a computer program

Description automatically generated

**Dependency Inversion Principle (DIP) - Princíp inverzie závislostí**

* **Definícia**: Moduly vyššej úrovne by nemali závisieť na moduloch nižšej úrovne. Obe by mali závisieť na abstrakciách. Okrem toho by abstrakcie nemali závisieť na detailoch, ale detaily by mali závisieť na abstrakciách.
* **Praktický príklad**: Ak máme aplikáciu, kde **BusinessLogicLayer** priamo závisí na **DataAccessLayer**, akúkoľvek zmenu v **DataAccessLayer** by mohla vyžadovať zmenu v **BusinessLogicLayer**. Podľa DIP by sme mali zaviesť abstraktné rozhranie pre **DataAccessLayer**, na ktorom by **BusinessLogicLayer** závisel, čím by sme izolovali vyššiu vrstvu od priamych zmien v nižšej vrstve.

- pred DIP aplikáciou:

A computer screen shot of code

Description automatically generated

- po DIP aplikácii:

A computer screen shot of a code

Description automatically generated

Tieto princípy, keď sú správne aplikované, vedú k návrhu softvéru, ktorý je robustnejší, ľahšie rozšíriteľný a udržateľný. Dôležité je však tiež vedieť, kedy a ako tieto princípy použiť, pretože nadmerné alebo nesprávne použitie môže viesť k zbytočnej zložitosti.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Na čo je dobrý Atomic Integer? (nepotrebuje využívať lock, mal by sa správať efektívnejšie ako synchronized block)**

**AtomicInteger** je trieda z balíčka **java.util.concurrent.atomic** v Jave, ktorá poskytuje možnosť vykonávať bezpečné operácie s celými číslami v multithreadovom prostredí bez potreby použitia **synchronized** blokov alebo metód. Táto trieda využíva nízkoúrovňové atomické operácie hardvéru, čo znamená, že operácie ako aktualizácia (napríklad inkrementácia, dekrementácia, nastavenie novej hodnoty) sú vykonávané ako jedna nedeliteľná (atomická) operácia. To zaisťuje, že v každom okamihu má premenná konzistentný stav, aj keď ju súčasne upravujú viaceré vlákna.

**Na čo je AtomicInteger dobrý?**

1. **Thread-Safe Operácie bez synchronized:** Umožňuje bezpečnú prácu s celočíselnými hodnotami v multithreadovom prostredí bez potreby obalovať operácie v **synchronized** blokoch alebo metódach, čo môže viesť k vyššej efektívnosti, najmä v prostrediach s vysokou konkurenciou.
2. **Vysoká Výkonnosť:** Keďže **AtomicInteger** využíva nízkoúrovňové atomické inštrukcie procesora, jeho operácie sú typicky rýchlejšie a efektívnejšie ako použitie blokov synchronizovaných na vyššej úrovni, ktoré môžu zaviesť nepotrebnú režiu a spomaliť vykonávanie programu.
3. **Jednoduchosť použitia:** Poskytuje jednoduché API pre bežné atomické operácie, ako sú **get()**, **set()**, **getAndIncrement()**, **getAndDecrement()**, **getAndAdd()**, **compareAndSet()**, atď., čo uľahčuje implementáciu bezpečných operácií s celými číslami bez hlbokého porozumenia mechanizmov synchronizácie.
4. **Podpora Lock-Free Programovania:** Umožňuje vytvárať efektívne lock-free algoritmy a dátové štruktúry, ktoré môžu zlepšiť celkovú reakčnú schopnosť a priepustnosť aplikácie v prostrediach s vysokým stupňom súbehu.

**Praktický Príklad**

Predstavte si situáciu, kde chcete mať počítadlo prístupov k nejakej zdroji, ktoré je bezpečne prístupné z viacerých vlákien:

A computer screen shot of a black screen

Description automatically generated

V tomto príklade **AtomicInteger** zabezpečuje, že každé vlákno bude vidieť aktuálnu hodnotu počítadla a že každá inkrementácia sa vykoná atomicky, čím sa predíde konfliktom a inkonzistenciám, ktoré by mohli nastať pri použití bežných premenných a synchronizácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JAVA Stream API - aké dva typy (intermediate a ...)?**  
V Java Stream API existujú dva hlavné typy operácií: intermediate (medzioperácie) a terminal (terminálne operácie).

**Intermediate Operácie (Medzioperácie)**

* **Charakteristika**: Sú len vykonané, keď sa vykoná terminálna operácia. Sú to teda len "prípravné" operácie, ktoré transformujú stream na iný stream. Tieto operácie sú lenavé (lazy), čo znamená, že nevykonávajú žiadnu prácu skutočne až do momentu, kedy je na stream aplikovaná terminálna operácia.
* **Príklady**: **filter()**, **map()**, **flatMap()**, **distinct()**, **sorted()**, **peek()**, **limit()**, **skip()** atď.

**Terminal Operácie (Terminálne operácie)**

* **Charakteristika**: Spustia vykonávanie všetkých medzioperácií na stream-e a vracajú výsledok. Po vykonaní terminálnej operácie sa stream stáva "spotrebovaným" a už nemôže byť ďalej používaný pre ďalšie operácie.
* **Príklady**: **forEach()**, **forEachOrdered()**, **toArray()**, **reduce()**, **collect()**, **min()**, **max()**, **count()**, **anyMatch()**, **allMatch()**, **noneMatch()**, **findFirst()**, **findAny()** atď.

Tieto dva typy operácií umožňujú vytvárať komplexné dátovej tokové transformácie v elegantnom a funkcionálnom štýle. Intermediate operácie umožňujú transformovať a filtrovať dáta, kým terminal operácie umožňujú vyhodnotiť výsledky týchto transformácií a filtrovaní.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Rozdiel medzi STATEMENT a PREPARE STATEMENT? Poznáš aj 3. typ STATEMENTU?**  
  
V Jave, keď pracujete s JDBC (Java Database Connectivity) na interakciu s databázou, môžete naraziť na tri hlavné typy **Statement** objektov, ktoré sa používajú na vykonávanie SQL príkazov: **Statement**, **PreparedStatement** a **CallableStatement**. Každý z týchto typov má svoje špecifické použitie a výhody:

**1. Statement**

* **Použitie**: Na vykonávanie jednoduchých SQL príkazov bez parametrov.
* **Charakteristika**: **Statement** objekty sú základným spôsobom, ako vykonávať SQL príkazy. Príkaz je definovaný ako reťazec a môže byť priamo vykonaný na databáze.
* **Nevýhody**: Keďže **Statement** objekty neumožňujú parametrizáciu, sú menej efektívne a môžu byť náchylné na SQL injekcie pri nesprávnom použití.

A computer screen with white text

Description automatically generated

**2. PreparedStatement**

* **Použitie**: Na vykonávanie SQL príkazov s jedným alebo viacerými vstupnými parametrami.
* **Charakteristika**: **PreparedStatement** objekty umožňujú definovať SQL príkazy s jedným alebo viacerými parametrami, ktoré môžu byť nastavené pred vykonaním príkazu. Tieto objekty zvyšujú výkon tým, že umožňujú databáze opätovne používať preložené SQL príkazy a znižujú riziko SQL injekcií.
* **Výhody**: Zvýšená bezpečnosť a možnosť opakovaného použitia, čo vedie k efektívnejšiemu vykonávaniu.

A screen shot of a computer

Description automatically generated

**3. CallableStatement**

* **Použitie**: Na vykonávanie uložených procedúr v databáze, ktoré môžu mať vstupné a výstupné parametre.
* **Charakteristika**: **CallableStatement** objekty sú používané na volanie uložených procedúr, ktoré sú definované v databáze. Tieto objekty umožňujú prácu s vstupnými a výstupnými parametrami, čím poskytujú flexibilný spôsob interakcie s komplexnejšími databázovými operáciami.
* **Výhody**: Umožňujú efektívne využitie databázových funkcií a procedúr, čo môže viesť k lepšej organizácii a výkonnosti databázového kódu.

Predpokladajme, že máme uloženú procedúru v databáze s názvom **increaseSalary** s dvoma parametrami: **employeeId** (vstupný parameter) a **newSalary** (výstupný parameter).

A black screen with white text

Description automatically generated

String sql = "{CALL increaseSalary(?, ?)}"; // SQL volanie uloženej procedúry

try (Connection conn = DriverManager.getConnection(url, username, password);

CallableStatement cstmt = conn.prepareCall(sql)) {

cstmt.setInt(1, 101); // Nastavenie vstupného parametra (employeeId)

cstmt.registerOutParameter(2, Types.DECIMAL); // Registrácia výstupného parametra (newSalary)

cstmt.execute();

BigDecimal newSalary = cstmt.getBigDecimal(2); // Získanie hodnoty výstupného parametra

System.out.println("Nový plat: " + newSalary);

} catch (SQLException e) {

e.printStackTrace();

}

A computer screen with white text

Description automatically generated

Rozdiel medzi **Statement** a **PreparedStatement** spočíva hlavne v bezpečnosti a efektívnosti. **PreparedStatement** je preferovaný kvôli jeho výhodám v týchto oblastiach. **CallableStatement** je špeciálne určený pre prácu s uloženými procedúrami, čo rozširuje možnosti interakcie s databázou nad rámec jednoduchých SQL príkazov.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**S čím sa nahradzuje UNSAFE?**

**Unsafe** je neoficiálna trieda v Sun Microsystems/Oracle JVM, ktorá poskytuje "backdoor" prístup k nízkoúrovňovým operáciám, ako je priama manipulácia s pamäťou, vytváranie inštancií tried bez volania konštruktorov a iné nebezpečné operácie, ktoré standardné Java API obvykle neumožňuje. Kvôli svojim schopnostiam a rizikám spojeným s jej používaním bola **Unsafe** vždy považovaná za internú API JDK a nie je súčasťou oficiálneho Java SE API. Toto znamená, že jej používanie môže spôsobiť problémy s prenositeľnosťou a kompatibilitou a môže byť zmenené alebo odstránené v budúcich verziách JDK bez varovania.

**S čím sa nahradzuje Unsafe?**

S uvedením modulového systému v Java 9 a ďalšími snahami o zabezpečenie Java platformy sa vývojári JDK snažia poskytnúť oficiálne API pre funkcie, ktoré boli predtým dostupné iba prostredníctvom **Unsafe**. Niektoré z týchto funkcií boli nahradené alebo majú byť nahradené bezpečnejšími alternatívami:

1. **VarHandle (Variable Handles)**: Predstavené v Java 9, **VarHandle** poskytujú bezpečný a oficiálny spôsob pre vykonávanie atomických operácií, čítanie a zápis do volatilných polí, a prácu s pamäťou na byte-level bez použitia **Unsafe**. **VarHandle** poskytujú podobnú funkčnosť s vylepšenou bezpečnosťou a prenositeľnosťou.
2. **Project Panama**: Tento projekt je zameraný na zlepšenie pripojenia medzi Javou a natívnym kódom, vrátane práce s pamäťou mimo Java heap. Cieľom Project Panama je nahradenie niektorých použití **Unsafe** pri práci s natívnou pamäťou a volaním natívných funkcií.
3. **Project Valhalla**: Zameraný na vylepšenie modelu dát v Jave, najmä pokiaľ ide o generiká a hodnotové typy (value types). Tento projekt by mohol poskytnúť bezpečnejšie a efektívnejšie alternatívy pre niektoré použitia **Unsafe**, najmä v oblastiach, kde sú požadované vysokovýkonné operácie s dátami.

**Čo to je Unsafe?**

**Unsafe** je trieda, ktorá poskytuje prístup k operáciám, ktoré môžu obísť bezpečnostné a dizajnové garantie Javy. Zahrnuje operácie ako priame čítanie a zápis do pamäti, vytváranie inštancií tried bez volania ich konštruktorov, a prístup k nízkoúrovňovým funkcionalitám operácií systému. Kvôli týmto schopnostiam a potenciálnym rizikám s nimi spojeným bola vždy považovaná za pokročilú a riskantnú API, ktorú by mali používať len skúsení vývojári a to s veľkou opatrnosťou.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------**Na akých princípoch fungujú ATOMIC?**  
Atomic operácie v programovaní a v kontexte multithreadingu sú základom pre bezpečnú prácu s dátami vo viacvláknových aplikáciách bez nutnosti explicitnej synchronizácie pomocou zámkov (locks). Tieto operácie sú "atomické" v tom zmysle, že každá operácia je vykonaná celá ako jeden nedeliteľný krok, čo znamená, že žiadne iné vlákno nemôže vidieť operáciu v "polovičnom" stave. Ak teda jedno vlákno vykonáva atomickú operáciu, iné vlákno uvidí buď stav pred vykonaním operácie, alebo stav po jej vykonaní, ale nikdy niečo medzi tým.

**Princípy fungovania atomic operácií**

1. **Nedeliteľnosť (Indivisibility):**
   * Atomic operácie sú nedeliteľné, čo znamená, že sa vykonávajú v jednom kroku bez možnosti prerušenia alebo pozorovania čiastočného vykonania iným vláknom. Toto je základný princíp, ktorý zabezpečuje konzistentnosť dát v konkurentnom prostredí.
2. **Viditeľnosť (Visibility):**
   * Zmeny vykonané atomickými operáciami sú okamžite viditeľné pre všetky ostatné vlákna. To zabezpečuje, že keď jedno vlákno aktualizuje hodnotu pomocou atomic operácie, všetky ostatné vlákna uvidia túto aktualizovanú hodnotu.
3. **Poradie (Ordering):**
   * Atomic operácie často zaručujú určitý stupeň usporiadania operácií. Napríklad v niektorých systémoch môže byť zaručené, že atomic operácie sú vykonávané v poradí, v akom boli požadované, čo pomáha predchádzať problémom súvisiacim s poradím prístupu k dátam.
4. **Compare-and-Swap (CAS):**
   * Mnohé implementácie atomických operácií využívajú techniku zvanú compare-and-swap. CAS operácia skontroluje hodnotu na určitej adrese a ak táto hodnota zodpovedá očakávanej hodnote, atomicky ju aktualizuje na novú hodnotu. Tento model umožňuje bezpečné aktualizácie hodnôt bez použitia zámkov.
5. **Lock-Free Programming:**
   * Využitím atomic operácií môžu byť vytvorené lock-free dátové štruktúry a algoritmy, ktoré zvyšujú priepustnosť a znížujú latenciu tým, že eliminujú potrebu blokovania vlákien.

**Implementácia v Jave**

V Jave, triedy v balíčku **java.util.concurrent.atomic**, ako sú **AtomicInteger**, **AtomicLong**, **AtomicReference**, a ďalšie, poskytujú atomic operácie pre základné typy a objekty. Tieto triedy využívajú efektívne nízkoúrovňové atomic operácie poskytované hardvérom alebo operačným systémom, čím zabezpečujú vysokú výkonnosť konkurentných operácií bez potreby synchronizácie na úrovni zámkov.

Atomic operácie sú kľúčovým nástrojom pre vývoj vysoko výkonných konkurentných aplikácií, kde je dôležitá bezpečnosť a efektívnosť prístupu k dátam medzi viacerými vláknami.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**Je STRING IMMUTABLE?**   
Áno, v Jave je trieda **String** immutable, čo znamená, že raz vytvorené reťazce (stringy) sa nedajú zmeniť. Akékoľvek operácie, ktoré vyzerajú, ako by menili obsah reťazca, v skutočnosti vytvárajú nový reťazec. Táto vlastnosť umožňuje, aby boli reťazce bezpečne používané v multithreadovom prostredí bez potreby dodatočnej synchronizácie, pretože nemôže dôjsť k zmenám stavu objektu **String** inými vláknami.

**Prečo sú reťazce v Jave immutable?**

1. **Bezpečnosť**: Immutability reťazcov umožňuje, aby mohli byť bezpečne používané ako kľúče v mapách a ako hodnoty v setoch bez obáv, že ich zmena ovplyvní integritu týchto dátových štruktúr.
2. **Optimalizácia**: JVM môže optimalizovať využívanie pamäte zdieľaním identických reťazcov medzi viacerými objektmi alebo oblasťami kódu. Toto je známe ako internovanie reťazcov.
3. **Thread-Safety**: Vďaka nemennosti sú reťazce prirodzene thread-safe, čo znamená, že môžu byť používané bez obáv z konkurenčných modifikácií medzi vláknami.

**Príklad**

Keď vykonávate operáciu, ktorá by sa zdala meniť reťazec, ako napríklad konkatenáciu, v skutočnosti sa vytvára nový objekt **String**:

A black background with white text

Description automatically generated

V tomto príklade, aj keď sa môže zdať, že **s1** je zmenený pridaním ", world!", v skutočnosti sa vytvára úplne nový reťazec **s2**, a pôvodný reťazec **s1** ostáva nezmenený.

Táto vlastnosť reťazcov v Jave má dôležité dôsledky pre návrh softvéru a môže ovplyvniť výkon aplikácie, najmä v situáciách, kde je potrebné vykonať veľké množstvo modifikácií reťazcov. V takých prípadoch je často efektívnejšie použiť triedy ako **StringBuilder** alebo **StringBuffer**, ktoré sú navrhnuté pre mutabilnú manipuláciu s reťazcami.Top of Form

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké napojenia/prístupy do dtb poznáš (entity manager, tripper statement, coller booster, jpa, jdbc...)**

Pri napojení na databázu v Jave existuje niekoľko prístupov a technológií. Niektoré z menovaných v otázke sú zrejme nesprávne interpretované alebo chybne napísané, tak ich skúsim opraviť a rozšíriť o správne informácie:

**1. JDBC (Java Database Connectivity)**

* **Popis**: Je to nízkoúrovňové API pre interakciu s rôznymi typmi databáz. Poskytuje metódy na vykonávanie SQL príkazov, spracovanie výsledkov dotazov a správu transakcií.
* **Použitie**: Priame vykonávanie SQL príkazov, ako sú **SELECT**, **INSERT**, **UPDATE**, a **DELETE**.

**2. JPA (Java Persistence API)**

* **Popis**: Je to špecifikácia, ktorá poskytuje rámec na mapovanie objektov do tabuliek databázy (ORM) a správu týchto objektov. JPA umožňuje pracovať s databázou na vyššej úrovni abstrakcie.
* **Použitie**: Správa entít (objektov) a ich persistentného stavu bez priameho používania SQL.

**3. Hibernate (Implementácia JPA)**

* **Popis**: Je to populárna implementácia JPA, ktorá poskytuje rozšírenú podporu pre mapovanie objektov a manipuláciu s dátami. Hibernate umožňuje efektívnejšiu prácu s databázou prostredníctvom cachovania, lazy loadingu a ďalších optimalizácií.
* **Použitie**: ORM, automatické generovanie SQL príkazov, správa relácií medzi objektmi.

**4. Spring Data JPA**

* **Popis**: Je to modul Spring Frameworku, ktorý zjednodušuje implementáciu databázových prístupových vrstiev poskytovaním podpory na úrovni repozitárov. Využíva JPA na zjednodušenie CRUD operácií a dotazovania.
* **Použitie**: Práca s databázovými entitami pomocou jednoduchých rozhraní bez nutnosti písať boilerplate kód.

**5. Entity Manager**

* **Popis**: Je súčasťou JPA a poskytuje API pre správu persistentných entít (objektov), ich životný cyklus, transakcie a vykonávanie dotazov.
* **Použitie**: Správa entít v kontexte JPA.

Pokiaľ ide o "tripper statement" a "coller booster", tieto termíny sa nezdajú byť štandardnými pojmami používanými v kontexte databázových prístupov v Jave. Je možné, že išlo o nesprávnu interpretáciu alebo chybné napísanie. Možno ste mali na mysli **Statement**, **PreparedStatement** a **CallableStatement**, ktoré sú súčasťou JDBC a slúžia na vykonávanie SQL príkazov, pričom **PreparedStatement** umožňuje parametrizované dotazy a **CallableStatement** je používaný na volanie uložených procedúr.

Každý z týchto prístupov má svoje výhody a nevýhody a výber závisí od konkrétnych potrieb aplikácie, požadovanej úrovne abstrakcie a komplexnosti manipulácie s dátami.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako by si to riešil v Springu? (JDBC??)**

Ak chcete v Spring frameworku používať prístup JDBC, môžete využiť **JdbcTemplate**, ktorý je navrhnutý tak, aby zjednodušil proces interakcie s databázou priamo prostredníctvom JDBC. **JdbcTemplate** automatizuje bežné úlohy, ako je otváranie a zatváranie spojení, beh dotazov, iterácia cez výsledky a výnimky, čím minimalizuje potrebu boilerplate kódu.

**Konfigurácia**

Predtým, než začnete používať **JdbcTemplate**, budete musieť nakonfigurovať **DataSource**. Toto môže byť jednoducho vykonané v konfiguračnej triede Springu.

A computer screen shot of a code

Description automatically generated

**Použitie JdbcTemplate**

Po nakonfigurovaní **DataSource** a **JdbcTemplate** môžete začať vykonávať databázové operácie. **JdbcTemplate** podporuje širokú škálu operácií, vrátane dotazovania, aktualizácií, vkladania a mazania.

Vykonávanie SELECT dotazov

Na vykonávanie SELECT dotazov a získanie výsledkov môžete použiť rôzne metódy, ako napríklad **queryForObject**, **queryForList**, alebo **query**.

**Príklad: Získanie jedného záznamu:**

A computer screen shot of a program code

Description automatically generated

Vykonávanie INSERT, UPDATE, DELETE

Na vykonávanie operácií, ktoré menia dáta (napr. INSERT, UPDATE, DELETE), môžete použiť metódu **update**.

**Príklad: Pridanie záznamu**

A computer screen shot of a black background

Description automatically generated

**JdbcTemplate** je mocný nástroj, ktorý výrazne zjednodušuje prácu s databázou v Spring aplikáciách, znižuje množstvo potrebného boilerplate kódu a zároveň poskytuje flexibilitu pre správu databázových operácií.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Implementacie JPA**

Java Persistence API (JPA) je štandardná špecifikácia v Jave pre mapovanie objektov na tabuľky v databáze, teda pre ORM (Object-Relational Mapping). JPA poskytuje abstraktnú vrstvu nad konkrétnou implementáciou ORM, umožňujúc vývojárom pracovať s databázami na vyššej úrovni abstrakcie. Hoci JPA definuje API a súbor funkcií pre ORM, samotnú funkcionalitu poskytujú konkrétne implementácie. Tu sú niektoré z najznámejších implementácií JPA:

**1. Hibernate**

* **Popis**: Hibernate je jednou z najpopulárnejších a najrozšírenejších implementácií JPA. Ponúka bohatú sadu funkcií nad rámec štandardného JPA, vrátane cachovania, lazy loadingu, a podpory pre rôzne dialekty SQL.
* **Výhody**: Vysoká flexibilita, silná komunitná podpora, rozsiahle možnosti konfigurácie a optimalizácie.
* **Použitie**: Vhodné pre komplexné aplikácie, kde je potrebná pokročilá kontrola nad mapovaním a správou databázy.

**2. EclipseLink**

* **Popis**: EclipseLink je referenčnou implementáciou JPA, ktorá bola pôvodne vyvinutá ako časť projektu TopLink od spoločnosti Oracle. Poskytuje podporu pre XML a JSON marshalling a je známa svojou výkonnosťou.
* **Výhody**: Dobrá integrácia s inými technológiami, ako je JAXB; vysoká výkonnosť a flexibilita.
* **Použitie**: Odporúčané pre projekty, kde je potrebná integrácia s rôznymi formátmi dát a technológiami.

**3. OpenJPA**

* **Popis**: OpenJPA je ďalšia implementácia JPA, ktorá bola vyvinutá pod záštitou Apache Software Foundation. Je navrhnutá tak, aby bola ľahko použiteľná a integrovateľná do Java EE a ľahkých Java SE aplikácií.
* **Výhody**: Dobre integrovaná s Apache projektmi, jednoduchá na použitie a konfiguráciu.
* **Použitie**: Vhodná pre aplikácie, kde je potrebná integrácia s ďalšími Apache projektmi alebo pre projekty hľadajúce jednoduchú implementáciu JPA.

**4. DataNucleus**

* **Popis**: DataNucleus poskytuje implementáciu JPA a tiež podporu pre JDO (Java Data Objects), čo je ďalšia špecifikácia pre prácu s persistentnými dátami v Jave. Je navrhnutý pre flexibilitu a modularitu.
* **Výhody**: Podpora pre rôzne formáty dát a úložiská (relačné databázy, NoSQL databázy, webové služby).
* **Použitie**: Ideálna pre projekty, ktoré vyžadujú prácu s rôznymi typmi úložísk alebo formátov dát.

Pri výbere implementácie JPA by ste mali zvážiť špecifické požiadavky vášho projektu, ako sú výkonnosť, kompatibilita s databázami, podpora komunity a dostupné funkcie. Každá z uvedených implementácií má svoje silné stránky a obmedzenia, takže výber by mal byť založený na konkrétnych potrebách a preferenciách.  
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Čo to je spring boot a ako funguje?**

Spring Boot je projekt v rámci širšieho ekosystému Spring, ktorý zjednodušuje proces vývoja nových Spring aplikácií prostredníctvom konvencie nad konfiguráciou. Cieľom Spring Boot je minimalizovať množstvo konfiguračného kódu, ktorý je potrebný na spustenie Spring aplikácie, a poskytnúť súbor štandardných konfigurácií pre rýchly štart a vývoj.

**Ako funguje Spring Boot?**

Spring Boot funguje na základe niekoľkých kľúčových princípov a funkcií:

1. **Automatická Konfigurácia**: Spring Boot automaticky konfiguruje vašu aplikáciu na základe pridaných závislostí v projekte. Napríklad, ak pridáte **spring-boot-starter-web**, Spring Boot automaticky nakonfiguruje Tomcat a Spring MVC tak, aby ste mohli začať vytvárať webové aplikácie bez potreby manuálnej konfigurácie.
2. **Štartovacie Závislosti (Starters)**: Spring Boot poskytuje širokú škálu štartovacích závislostí (starters), ktoré agregujú bežne používané závislosti do jednoducho použiteľných skupín. Tieto starters zjednodušujú proces pridávania komponentov do vašej aplikácie, ako sú webové servery, šablónovacie nástroje, bezpečnosť a ďalšie.
3. **Bezproblémové Spravovanie Závislostí**: Spring Boot spravuje verzie použitých knižníc, aby boli kompatibilné medzi sebou, čím eliminuje potrebu manuálneho spravovania kompatibilných verzií.
4. **Podpora pre Rôzne Typy Aplikácií**: S Spring Boot môžete vytvárať webové aplikácie, RESTful služby, mikroslužby, batch procesy, a ďalšie typy aplikácií s minimálnym nastavením.
5. **Vstavaný Server**: Spring Boot aplikácie môžu byť spustené ako samostatné Java aplikácie, ktoré obsahujú vstavaný webový server (napr. Tomcat, Jetty alebo Undertow), čím sa eliminuje potreba nasadzovať aplikácie do externého webového servera.
6. **Aktuátory (Actuators)**: Poskytujú životné informácie o vašej aplikácii (zdravie, metriky, dump konfigurácie, atď.) prostredníctvom HTTP endpointov alebo JMX, čo uľahčuje monitorovanie a správu aplikácií v produkčnom prostredí.

**Spustenie Spring Boot Aplikácie**

Základ Spring Boot aplikácie môže vyzerať veľmi jednoducho, napríklad:

A black screen with white text

Description automatically generated

Anotácia **@SpringBootApplication** je pohodlná anotácia, ktorá zahŕňa **@Configuration**, **@EnableAutoConfiguration** a **@ComponentScan** anotácie s ich predvolenými atribútmi.

Spring Boot tak poskytuje rýchly a jednoduchý spôsob, ako vytvárať Spring aplikácie, znižuje potrebu manuálnej konfigurácie a umožňuje vývojárom sústrediť sa na business logiku aplikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Anotácia conditional ON a ďalšie...**

Anotácie podmieneného spustenia (Conditional annotations) v Spring Boot umožňujú určitú konfiguráciu alebo beans načítať alebo inicializovať len vtedy, keď sú splnené špecifiké podmienky. Tieto anotácie sú veľmi užitočné pre vytváranie flexibilných a konfigurovateľných aplikácií, ktoré sa môžu prispôsobiť rôznym prostrediam alebo konfiguráciám bez zmeny kódu. Tu je prehľad niektorých z najpoužívanejších anotácií podmieneného spustenia v Spring Boot:

**1. @ConditionalOnBean**

* **Popis**: Konfigurácia sa načíta len vtedy, keď v aplikácií existuje bean definovaný triedou alebo menom.
* **Použitie**: Používa sa, keď chcete inicializovať bean len v prípade, že už existuje iný špecifikovaný bean.

**2. @ConditionalOnMissingBean**

* **Popis**: Konfigurácia sa načíta len vtedy, keď v aplikácií neexistuje bean definovaný triedou alebo menom.
* **Použitie**: Používa sa na zabezpečenie, že sa bean inicializuje len ak ešte nebol definovaný iný konkrétny bean.

**3. @ConditionalOnClass**

* **Popis**: Konfigurácia sa načíta, ak sú na classpath-e prítomné špecifikované triedy.
* **Použitie**: Toto je užitočné, napríklad, pre konfigurácie, ktoré sú závislé od prítomnosti knižníc tretích strán.

**4. @ConditionalOnMissingClass**

* **Popis**: Konfigurácia sa načíta, ak špecifikované triedy nie sú na classpath-e.
* **Použitie**: Môže byť použité na alternatívne konfigurácie v prípade, že niektorá knižnica nie je dostupná.

**5. @ConditionalOnProperty**

* **Popis**: Konfigurácia sa načíta na základe prítomnosti a hodnoty špecifikovanej vlastnosti v **application.properties** alebo **application.yml**.
* **Použitie**: Veľmi užitočné pre zapínanie/vypínanie funkcií aplikácie na základe konfigurácie.

**6. @ConditionalOnResource**

* **Popis**: Konfigurácia sa načíta, ak je dostupný špecifikovaný zdroj (resource), napríklad súbor v classpath.
* **Použitie**: Môže byť použité na načítanie konfigurácie len ak existujú špecifické zdroje.

**7. @ConditionalOnWebApplication a @ConditionalOnNotWebApplication**

* **Popis**: Konfigurácia sa načíta len vtedy, keď aplikácia je (alebo nie je) webová aplikácia.
* **Použitie**: Umožňuje rozlišovať konfiguráciu pre webové a ne-webové aplikácie.

Tieto anotácie umožňujú veľmi flexibilné konfigurácie a sú základným stavebným kameňom pre "auto-configurations" v Spring Boot, ktoré automaticky konfigurujú aplikáciu na základe jej prostredia a závislostí.

**Na čo slúžia podmienené anotácie?**

**Dynamická Konfigurácia:** Umožňujú aplikácii prispôsobiť sa rôznym prostrediam alebo konfiguráciám bez potreby zmeny kódu. Napríklad môžete mať komponentu, ktorá sa má aktivovať len v produkčnom prostredí alebo keď je dostupná určitá databáza.

**Minimalizácia Závislostí:** Pomáhajú predchádzať vytváraniu nežiaducich alebo nepotrebných beans v kontexte aplikácie, čo môže zlepšiť štartovací čas a výkon aplikácie.

**Flexibilita:** Poskytujú vývojárom nástroje na vytváranie konfigurovateľných a modulárnych aplikácií, ktoré sa dajú jednoducho prispôsobiť na základe externých faktorov, ako sú konfiguračné súbory, systémové vlastnosti, prítomnosť knižníc atď.

**Kedy ich používať?**

**1. Prispôsobenie Konfigurácie:** Keď potrebujete prispôsobiť konfiguráciu aplikácie na základe prostredia, v ktorom aplikácia beží. Napríklad, môžete chcieť použiť rôzne konfigurácie pre vývojové, testovacie a produkčné prostredie.

**2. Voliteľné Závislosti:** Keď chcete, aby bola vaša aplikácia flexibilná a schopná pracovať aj v prípade, že niektoré knižnice nie sú k dispozícii. Môžete definovať komponenty, ktoré sa aktivujú len v prítomnosti týchto knižníc.

**3. Funkcionalita Na Požiadanie:** Ak má vaša aplikácia moduly alebo funkcie, ktoré by mali byť dostupné len na základe špecifických konfiguračných nastavení alebo systémových vlastností.

**4. Rozlišovanie medzi Webovými a Ne-webovými Aplikáciami:** Ak vyvíjate knižnicu alebo modul, ktorý má fungovať inak (alebo poskytovať rozdielne komponenty) v závislosti od toho, či je použitý vo webovej aplikácii alebo v aplikácii bez webového kontextu.

**Príklad použitia**

Predstavte si, že vyvíjate aplikáciu, ktorá by mala používať cache len v produkčnom prostredí. Môžete to jednoducho konfigurovať pomocou **@ConditionalOnProperty** anotácie:

A computer screen with white text

Description automatically generated

V tomto príklade sa **CachingConfig** konfigurácia aplikuje len vtedy, keď je vlastnosť **caching.enabled** nastavená na **true**.

Podmienené anotácie teda slúžia ako mocný nástroj na dynamickú konfiguráciu Spring aplikácií, umožňujú vývojárom vytvárať flexibilné a efektívne aplikácie prispôsobené rôznym potrebám a prostrediam.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Typy beanov v springu**  
V kontexte Spring Frameworku, termín "bean" označuje objekt, ktorý je inštanciou triedy spravovaný Spring IoC (Inversion of Control) kontajnerom. Spring beans sú základnými stavebnými kameňmi každej Spring aplikácie. Tieto objekty sú vytvorené, spravované, a konfigurované Spring kontajnerom a môžu byť vkladané do iných objektov ako závislosti, čo umožňuje ľahkú a flexibilnú modulárnu architektúru.

**Kľúčové aspekty Spring beans:**

**1. Správa životného cyklu:** Spring kontajner zodpovedá za celý životný cyklus beanu od jeho vytvorenia až po zničenie. Toto zahŕňa inicializáciu, nastavenie závislostí a vykonanie akýchkoľvek potrebných zatváracích operácií.

**2. Injektovanie závislostí:** Jednou z hlavných funkcií Spring IoC kontajnera je injektovanie závislostí do beanov. Toto môže byť vykonané cez konštruktory, setter metódy alebo priamo cez polia. Cieľom je znížiť závislosti medzi komponentami a zvýšiť tak modulárnosť a testovateľnosť kódu.

**3. Deklaratívna konfigurácia:** Bean môže byť konfigurovaný deklaratívne, buď pomocou XML konfiguračných súborov, anotácií alebo Java konfiguračnej triedy. Táto flexibilita umožňuje vývojárom zvoliť metódu konfigurácie, ktorá najlepšie vyhovuje ich projektu.

**4. Rozsah (Scope):** Každý bean môže mať definovaný rozsah, ktorý určuje, ako sa objekt správa v rámci aplikácie. Napríklad, bean so scope "singleton" existuje ako jediná inštancia v celom Spring kontajneri, zatiaľ čo bean so scope "prototype" generuje novú inštanciu pri každom požiadavku na bean.

**Ako sa beany definujú:**

Beany môžu byť definované rôznymi spôsobmi:

* **Anotácie:** **@Component**, **@Service**, **@Repository**, a **@Controller** sú špeciálne typy anotácií, ktoré automaticky označujú triedu ako Spring bean pri skenovaní tried Spring kontajnerom.
* **Java konfiguračné triedy:** Pomocou **@Bean** anotácie v metódach konfiguračnej triedy označenej **@Configuration**.
* **XML konfigurácia:** Definovanie beanov v XML súbore pomocou **<bean>** elementov.

Spring beans a IoC kontajner sú základom pre vytváranie udržateľných a flexibilných Java aplikácií s vysokou úrovňou dekompozície a znovupoužiteľnosti kódu.

V Spring frameworku môžeme rozlišovať rôzne typy beanov z hľadiska ich rozsahu (scope), životného cyklu a spôsobu vytvárania. Tu je prehľad základných typov beanov a ich charakteristík:

**1. Singleton (Predvolený rozsah)**

* **Charakteristika**: Spring kontajner vytvorí a udržiava presne jednu inštanciu beanu na celý kontajner. Tento typ je predvoleným rozsahom pre Spring beany.
* **Použitie**: Používa sa pre stavy bez stavu alebo zdieľané služby, ktoré sú počas životného cyklu aplikácie konštantné.

**2. Prototype**

* **Charakteristika**: Každé získanie beanu z kontajnera (napríklad cez **getBean** metódu) vytvorí novú inštanciu daného beanu.
* **Použitie**: Vhodné pre stavy, kde je potrebná nová inštancia pre každé použitie.

**3. Request**

* **Charakteristika**: Bean je vytvorený nový pre každú HTTP požiadavku. Tento rozsah je použiteľný len v kontexte webovej aplikácie.
* **Použitie**: Užitočné pre dáta špecifické pre danú požiadavku, ako sú užívateľské preferencie v session.

**4. Session**

* **Charakteristika**: Bean je vytvorený nový pre každú HTTP session. Tento rozsah je tiež použiteľný len v kontexte webovej aplikácie.
* **Použitie**: Používa sa na uchovávanie informácií špecifických pre session užívateľa.

**5. Application**

* **Charakteristika**: Bean je vytvorený raz pre celý ServletContext, teda je zdieľaný medzi všetkými užívateľskými sessionami v aplikácii.
* **Použitie**: Vhodné pre globálne nastavenia alebo dáta, ktoré majú byť zdieľané naprieč celou aplikáciou.

**6. WebSocket**

* **Charakteristika**: Bean je vytvorený pre životnosť WebSocket session. Tento rozsah je špecifický pre aplikácie, ktoré používajú WebSocket.
* **Použitie**: Užitočné pre správu stavu v rámci WebSocket komunikácie.

**Ako definovať rozsah beanu**

Rozsah beanu môžete definovať pomocou **@Scope** anotácie pri deklarácii beanu. Napríklad:

A screen shot of a computer program

Description automatically generated

V tomto príklade **MyPrototypeBean** má metódu **displayMessage()**, ktorá vypíše hodnotu svojho vnútorného stavu **message**. Keďže tento bean je označený ako prototype, pre každé jeho získanie zo Spring kontextu bude vytvorená nová inštancia, a teda volanie metódy **displayMessage()** na rôznych inštanciách bude ukazovať rovnakú správu, ale z rôznych objektov.

Pre demonštráciu, ako pracovať s týmto beanom, tu je príklad, ako by ste mohli získať a použiť **MyPrototypeBean** v Spring aplikácii:

A computer screen shot of a program

Description automatically generated

V tejto ukážke **BeanConsumer** používa Spring **ApplicationContext** na získanie inštancií **MyPrototypeBean**. Keďže **MyPrototypeBean** je definovaný ako prototype, **beanOne** a **beanTwo** sú dve odlišné inštancie.  
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**Špecifikátory prístupu v JAVA**

V Jave špecifikátory prístupu (alebo modifikátory prístupu) určujú úroveň prístupu k triedam, metódam, konštruktorom a členským premenným. Existujú štyri základné špecifikátory prístupu:

**1. public**

* **Popis**: Člen označený ako **public** je prístupný z akéhokoľvek iného miesta v programe, bez ohľadu na balíček, v ktorom sa nachádza.
* **Použitie**: Vhodný pre metódy a premenné, ktoré majú byť prístupné z iných tried a balíčkov.

**2. protected**

* **Popis**: Člen označený ako **protected** je prístupný v rámci svojej triedy, v triedach v rovnakom balíčku, a v triedach, ktoré sú potomkami tejto triedy (dedičnosť), aj keď sú v inom balíčku.
* **Použitie**: Používa sa, keď chcete obmedziť prístup k členom tak, aby boli prístupné len pre podtriedy.

**3. Bez špecifikátora (predvolený prístup, package-private)**

* **Popis**: Členy bez špecifikátora prístupu sú prístupné len v rámci tried v rovnakom balíčku. Tento prístup sa niekedy nazýva aj "package-private".
* **Použitie**: Vhodný pre metódy a premenné, ktoré by mali byť prístupné len v rámci balíčku.

**4. private**

* **Popis**: Člen označený ako **private** je prístupný len v rámci triedy, v ktorej je definovaný.
* **Použitie**: Používa sa pre metódy a premenné, ktoré by nemali byť prístupné zvonka svojej triedy. Ideálne pre skrytie vnútornej implementácie triedy.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako by si zadefinoval bean?**  
Bean v kontexte Spring Frameworku možno definovať ako objekt, ktorý je inštancovaný, zostavený a spravovaný Spring IoC (Inversion of Control) kontajnerom. Bean je základnou jednotkou kompozície pre aplikácie postavené na Springu a môže predstavovať dátový model, službu, komponentu alebo akýkoľvek iný objekt, ktorý vyžaduje logiku na spracovanie alebo uchovávanie stavu.

**Kľúčové aspekty beanov:**

* **Spravované IoC Kontajnerom**: Objekty (beany) sú vytvárané a spravované IoC kontajnerom, ktorý riadi ich životný cyklus a závislosti. Tento prístup umožňuje oddelenie konfigurácie a špecifikácií závislostí od skutočnej implementácie kódu.
* **Konfigurácia**: Beany môžu byť konfigurované deklaratívne pomocou XML, anotácií alebo Java konfiguračných tried, čo poskytuje flexibilitu pri definovaní a spravovaní závislostí medzi objektami.
* **Životný cyklus**: Spring poskytuje podporu pre správu celého životného cyklu beanov, vrátane vytvárania, inicializácie, používania a zničenia objektov.
* **Závislosti**: Spring IoC kontajner automaticky riadi závislosti medzi beanmi, čo umožňuje ľahkú a efektívnu injekciu závislostí a znižuje potrebu manuálneho kódovania na získavanie a spojovanie objektov.
* **Rozsahy (Scopes)**: Beany môžu mať rôzne rozsahy, ako sú singleton (predvolený), prototype, request, session a ďalšie, ktoré určujú, ako a kedy sú objekty vytvárané a spravované v rámci aplikácie.

**Príklad definície beanu pomocou anotácie:**

A black screen with white text

Description automatically generated

V tomto príklade, **@Component** anotácia označuje triedu **ExampleService** ako Spring bean, ktorý má byť automaticky detegovaný a registrovaný Spring kontajnerom pri skenovaní balíčkov.

Definícia beanu v Spring teda umožňuje vývojárom vytvárať flexibilné, modulárne a ľahko testovateľné aplikácie s vysokou úrovňou abstrakcie a konfigurovateľnosti.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako by si zadefinoval anotácie?**

Anotácie v Jave sú formou metadát, ktoré poskytujú dáta o programe, ale nie sú súčasťou samotného programu. Sú to deklarácie, ktoré sa píšu v kóde a môžu byť použité na triedy, metódy, premenné, parametre a balíčky. Anotácie poskytujú spôsob, ako pridať informácie k programovým prvkám deklaratívnym spôsobom, čo môže byť neskôr využité kompilátorom Javy, Java virtuálnou mašinou (JVM), frameworkmi alebo nástrojmi v čase behu alebo v čase kompilácie.

**Kľúčové aspekty anotácií:**

* **Typizované Metadáta**: Anotácie poskytujú štruktúrovaný a typizovaný spôsob, ako pridať metadáta k programovým prvkám.
* **Zjednodušenie Kódu**: Môžu zjednodušiť kód tým, že odstraňujú potrebu explicitného konfiguračného kódu. Napríklad, namiesto používania XML súborov na konfiguráciu môžu anotácie poskytnúť rovnaké informácie priamo v kóde.
* **Rozšírenie Funkcionality**: Mnohé Java frameworky a knižnice, ako Spring alebo Hibernate, používajú anotácie na rozšírenie funkcionality, umožňujú injekciu závislostí, definovanie ORM mapovania, konfiguráciu bezpečnosti a viac.
* **Kompilátor a Čas Behu**: Anotácie môžu byť spracované buď v čase kompilácie pomocou anotačných procesorov, alebo v čase behu pomocou reflexie.

**Príklady použitia anotácií:** A screenshot of a computer

Description automatically generated

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Akou anotáciou by si si zadefinoval API?**

Pre definovanie API v Jave, najmä v kontexte vytvárania webových služieb alebo RESTful API, je bežne používaný Spring Framework, resp. jeho rozšírenie Spring Boot, ktoré poskytuje súbor anotácií špecificky navrhnutých pre tento účel. Medzi základné anotácie, ktoré sa používajú na definovanie endpointov API, patria:

**1. @RestController**

Táto anotácia je používaná na triede a kombinuje funkcionalitu **@Controller** a **@ResponseBody**. Označuje triedu ako controller, ktorý je pripravený na spracovanie HTTP požiadaviek a automaticky serializuje návratové hodnoty metód do JSON alebo XML.:

**@RestController**

**public class MyApiController {**

**// definícia endpointov**

**}**  
  
**2. @RequestMapping (a špecifické varianty ako @GetMapping, @PostMapping, atď.)**

Tieto anotácie sú používané na metódach v rámci **@RestController** triedy na mapovanie HTTP požiadaviek na konkrétne metódy. **@RequestMapping** umožňuje špecifikovať HTTP metódu (GET, POST, DELETE, PUT), cestu (URL) a ďalšie parametre požiadavky. Pre jednoduchosť a lepšiu čitateľnosť kódu sa často používajú špecifické varianty ako **@GetMapping** pre GET požiadavky, **@PostMapping** pre POST požiadavky, a pod.:

**@GetMapping("/users")**

**public List<User> getAllUsers() {**

**// vráti zoznam užívateľov**

**}**

**@PostMapping("/users")**

**public User createUser(@RequestBody User user) {**

**// vytvorí nového užívateľa**

**}**

**Príklad definície jednoduchého RESTful API:**

**@RestController**

**@RequestMapping("/api")**

**public class ExampleApiController {**

**@GetMapping("/hello")**

**public String sayHello() {**

**return "Hello, World!";**

**}**

**@PostMapping("/echo")**

**public String echo(@RequestBody String message) {**

**return message;**

**}**

**}**

V tomto príklade, **@RestController** označuje **ExampleApiController** ako controller pre spracovanie HTTP požiadaviek, **@RequestMapping("/api")** definuje základnú cestu pre všetky metódy v controlleri, **@GetMapping("/hello")** mapuje GET požiadavku na **/api/hello** na metódu **sayHello**, a **@PostMapping("/echo")** mapuje POST požiadavku na **/api/echo** na metódu **echo**, ktorá vracia späť prijatú správu.

Použitím týchto anotácií môžete efektívne vytvárať a definovať RESTful API v Spring aplikáciách.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**Ako si zadefinujes REPOSITORY BEAN?**

V Spring Frameworku, na definovanie repository beanu, ktorý slúži ako most medzi datovou vrstvou aplikácie a jej biznis logikou, sa typicky používa anotácia **@Repository**. Táto anotácia je špeciálnym prípadom stereotypu **@Component** anotácie, ktorá označuje triedu ako komponentu Springu zodpovednú za prístup k dátam. Použitie **@Repository** nielenže označuje triedu ako súčasť datovej vrstvy, ale aj automaticky zapúšťa podporu pre prekladanie výnimiek závislých od konkrétneho úložiska dát do konzistentných Spring DataAccessExceptions.

**Definovanie Repository Beanu**

**1. Vytvorenie rozhrania repository:**

Najskôr definujete rozhranie repository, ktoré rozširuje jedno z repozitárových rozhraní poskytovaných Spring Data, ako napríklad **CrudRepository** alebo **JpaRepository**. Tieto rozhrania poskytujú základné CRUD (Create, Read, Update, Delete) operácie nad entitami bez potreby implementácie metód:

A screen shot of a computer program

Description automatically generated

V tomto príklade **UserRepository** rozširuje **JpaRepository**, ktoré poskytuje bohatú sadu CRUD operácií pre entitu **User** identifikovanú kľúčom typu **Long**. Anotácia **@Repository** nie je v tomto prípade technicky nevyhnutná, keďže Spring Data automaticky deteguje repozitáre a registruje ich ako beany, ale môže pomôcť s prehľadnosťou kódu a automatickým prekladom výnimiek.

**2. Použitie repository v servisnej vrstve:**

Následne môžete injektovať vaše repository do servisnej vrstvy a používať ho na manipuláciu s entitami:

A computer screen shot of a program code

Description automatically generated

Tento príklad ukazuje, ako môžete využívať **UserRepository** v **UserService** na vykonávanie databázových operácií, ako je získavanie všetkých používateľov. Vďaka Spring Data a anotácii **@Service** pre servisnú vrstvu sa celý proces práce s dátami stáva efektívnejším a minimalizuje sa potrebný boilerplate kód.

Vďaka takémuto prístupu k definícii a používaniu repository beanov Spring umožňuje vývojárom efektívne a elegantne pracovať s datovou vrstvou aplikácie, zatiaľ čo automatizuje mnoho bežných úloh spojených s prístupom k dátam.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**(Bonusova) Tri hlavne vrstvy v aplikaciach**  
V typickej aplikácii môžeme rozlíšiť tri hlavné vrstvy, ktoré spolu úzko súvisia a navzájom na seba nadväzujú. Tieto vrstvy sú:

1. **Dátová vrstva (Data Layer)**
2. **Biznis logika (Business Logic Layer)**
3. **Prezentačná vrstva (Presentation Layer)**

**1. Dátová vrstva (Data Layer)**

Dátová vrstva je základom aplikácie a zodpovedá za ukladanie, načítavanie a spravovanie dát. Táto vrstva komunikuje s databázou alebo inými zdrojmi dát (napr. externé API, súbory) a poskytuje rozhranie pre vyššie vrstvy na prístup a manipuláciu s dátami. V Spring aplikáciách sa pre prácu s dátovou vrstvou často používajú Spring Data JPA, JDBC Template alebo iné technológie ORM.

**2. Biznis logika (Business Logic Layer)**

Biznis logika predstavuje jadro aplikácie a zahŕňa pravidlá a algoritmy, ktoré definujú ako aplikácia spracováva dáta a vykonáva operácie. Táto vrstva transformuje dáta získané z dátovej vrstvy na informácie, ktoré sú užitočné pre užívateľa alebo iné systémy. Biznis logika by mala byť oddelená od prezentačnej a dátovej vrstvy, čo umožňuje jej ľahkú zmenu bez ovplyvnenia ostatných častí aplikácie.

**3. Prezentačná vrstva (Presentation Layer)**

Prezentačná vrstva je rozhranie medzi aplikáciou a jej užívateľmi alebo klientmi. Je zodpovedná za zobrazenie dát užívateľovi a spracovanie užívateľských interakcií. V webových aplikáciách zahŕňa HTML stránky, CSS štýly a JavaScript, zatiaľ čo v desktopových alebo mobilných aplikáciách zahŕňa užívateľské rozhrania vytvorené pomocou špecifických nástrojov a knižníc. Spring MVC je príkladom frameworku, ktorý sa používa na vytváranie prezentačnej vrstvy v Spring aplikáciách.

**Ako na seba vrstvy nadväzujú:**

* **Dátová vrstva** poskytuje abstrakciu nad zdrojmi dát a umožňuje biznis logike pracovať s dátami bez potreby zaoberať sa konkrétnymi detailmi ukladania dát.
* **Biznis logika** používa dáta z dátovej vrstvy na vykonávanie operácií, vyhodnotenie pravidiel a generovanie výstupov, ktoré majú hodnotu pre užívateľa alebo iný systém.
* **Prezentačná vrstva** preberá výstupy z biznis logiky a prezentuje ich užívateľovi v pochopiteľnej forme, zároveň umožňuje užívateľom interagovať s aplikáciou.

Táto trojvrstvová architektúra umožňuje oddelenie zodpovedností, čo vedie k lepšej organizácii kódu, uľahčuje testovanie a údržbu aplikácie a zvyšuje jej flexibilitu a rozširiteľnosť.

Top of Form

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako si zadefinujes REST service? (neviem ako kokot myslel tuto otazku …..)

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**BEAN SCOPE?**  
V kontexte Java a Spring framework, "Bean Scope" sa týka životného cyklu a viditeľnosti bean (objektu) spravovaného Spring kontajnerom. Spring definuje rôzne oblasti (scopes) pre beany, z ktorých každá určuje, ako a kedy je vytvorená nová inštancia bean. Tu sú základné oblasti používané v Spring:

1. **Singleton**: Predvolený scope. Pre každý Spring kontajner existuje práve jedna inštancia bean. Singleton scope znamená, že bean je vytvorený práve raz na celý kontajner a všetky požiadavky na daný bean dostanú tú istú inštanciu.
2. **Prototype**: Pre každú požiadavku na bean sa vytvorí nová inštancia. Tento scope je užitočný, keď potrebujete, aby každé použitie bean bolo čerstvo vytvorenou inštanciou.
3. **Request**: Bean je vytvorený na každú HTTP požiadavku. Tento scope je špecifický pre webové aplikácie a znamená, že nová inštancia bean je vytvorená pre každú HTTP požiadavku.
4. **Session**: Bean je vytvorený pre HTTP session. Tento scope je tiež špecifický pre webové aplikácie, pričom nová inštancia bean je vytvorená pre každú HTTP session.
5. **Global Session**: Bean je vytvorený pre globálnu HTTP session. Tento scope je používaný v portletových prostrediach a znamená, že bean je viazaný na globálnu portletovú session.
6. **Application**: Bean je vytvorený na úrovni servlet contextu, čo znamená, že je zdieľaný medzi všetkými užívateľskými sessionami v celej aplikácii.

Výber správneho scope pre bean je dôležitý, pretože to ovplyvňuje, ako sa s beanom pracuje v rámci aplikácie, ako je spravovaná jeho životnosť a ako sú zdieľané dáta medzi rôznymi komponentmi aplikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Rozdiel medzi SINGLETON a APPLICATION?**  
Rozdiel medzi **Singleton** a **Application** scope v kontexte Spring frameworku je založený na ich použití a správaní sa v rámci aplikácie:

1. **Singleton Scope**:
   * **Definícia**: Singleton scope znamená, že Spring IoC (Inversion of Control) kontajner vytvorí a spravuje práve jednu inštanciu beanu pre celý kontajner bez ohľadu na počet požiadaviek.
   * **Použitie**: Je to predvolený scope pre Spring beany. Singleton beany sú ideálne pre služby, ktoré neuchovávajú stav špecifický pre jednotlivé požiadavky alebo užívateľské sessiony, napríklad služby pre prístup k databáze, business služby atď.
   * **Životný cyklus**: Existuje len jedna inštancia beanu na Spring IoC kontajner, ktorá sa znovu používa pre všetky požiadavky a operácie v rámci aplikácie.
2. **Application Scope**:
   * **Definícia**: Application scope v Springu znamená, že bean je viazaný na životný cyklus **ServletContext**. To znamená, že bean je globálne dostupný v rámci celej webovej aplikácie, podobne ako globálna premenná v tradicionálnej webovej aplikácii.
   * **Použitie**: Application scoped beany sú užitočné, keď potrebujete zdieľať stav alebo informácie na úrovni celej aplikácie, napríklad konfiguračné nastavenia, ktoré sú spoločné pre všetky komponenty a operácie v aplikácii.
   * **Životný cyklus**: Inštancia beanu je vytvorená raz a je zdieľaná medzi všetkými komponentmi a operáciami v rámci aplikácie. Je dostupná počas celého životného cyklu **ServletContext**, čo znamená, že existuje od spustenia aplikácie až do jej zastavenia.

**Hlavný rozdiel** medzi nimi je v oblasti ich platnosti a dostupnosti. Zatiaľ čo Singleton scope je viazaný na konkrétny Spring IoC kontajner a je predvoleným scope pre beany v Spring aplikáciách, Application scope rozširuje dostupnosť beanu na úroveň celej webovej aplikácie, čo umožňuje zdieľanie stavu alebo informácií medzi rôznymi časťami aplikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké microservisové patterny (okrem SAGA) poznáš? (OUTBOX)**  
V architektúre mikroslužieb sa používa viacero vzorov (patternov) na riešenie rôznych problémov, ktoré prichádzajú s distribuovanými systémami. Okrem SAGA patternu, ktorý je známy pre správu transakcií medzi mikroslužbami, tu sú niektoré ďalšie užitočné vzory:

1. **API Gateway**:
   * Slúži ako jednotný vstupný bod pre rôzne mikroslužby. Uľahčuje klientom prístup k funkcionalitám systému tým, že agreguje rôzne služby do jediného API.
2. **Circuit Breaker**:
   * Pomáha zvládať zlyhania v distribuovaných systémoch. Ak služba zlyháva alebo je príliš pomalá, obvod sa "rozpojí" a volanie sa rýchlo preruší, čím sa zabraňuje ďalšiemu zhoršovaniu výkonnosti alebo zlyhaniu.
3. **Outbox Pattern**:
   * Používa sa na zabezpečenie spoľahlivého doručovania správ medzi mikroslužbami. Zapisuje výstupné správy do databázovej tabuľky (outbox), odkiaľ sa neskôr asynchrónne odosielajú do správneho cieľa. Tento vzor pomáha riešiť problémy s konzistenciou a integritou dát v distribuovaných transakciách.
4. **Event Sourcing**:
   * Uchováva zmeny stavu aplikácie ako sériu udalostí. Namiesto zápisu priameho stavu objektu do databázy, zaznamenáva sa postupnosť udalostí, ktoré viedli k jeho súčasnému stavu. Tento prístup umožňuje jednoduché vrátenie zmien a komplexné analýzy.
5. **CQRS (Command Query Responsibility Segregation)**:
   * Oddeluje čítanie dát (queries) od ich aktualizácie (commands), čo umožňuje optimalizáciu, škálovateľnosť a bezpečnosť pre každú operáciu zvlášť.
6. **Service Discovery**:
   * Umožňuje mikroslužbám dynamicky zisťovať a komunikovať s inými službami v systéme bez pevného kódovania ich umiestnenia. To zjednodušuje konfiguráciu a zvyšuje flexibilitu v cloudových prostrediach.
7. **Sidecar Pattern**:
   * Pridáva pomocné služby, ako je monitorovanie, logovanie, konfigurácia, k hlavnému kontajneru aplikácie, čím umožňuje izoláciu a nezávislú správu týchto cross-cutting concerns.
8. **Bulkhead Pattern**:
   * Izoluje elementy aplikácie do oddelených oblastí (podobne ako vodotesné priehradky na lodi), aby zlyhanie v jednej časti nemalo vplyv na ostatné časti. Tento vzor pomáha zlepšiť odolnosť systému proti zlyhaniam.

Tieto vzory adresujú rôzne aspekty vývoja a správy mikroslužieb, vrátane komunikácie, odolnosti proti zlyhaniam, správy transakcií a konzistencie dát.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Service mash? (i-styles?) - poznáš? (nieco s kubernetis)**

"Service Mesh" je termín používaný v architektúre mikroslužieb na označenie infraštruktúrnej vrstvy, ktorá zabezpečuje správu komunikácie medzi službami (mikroslužbami) v cloudových aplikáciách. Service mesh poskytuje spôsob, ako riadiť ako mikroslužby medzi sebou komunikujú, vrátane aspektov ako sú discovery služieb, load balancing, šifrovanie, autentifikácia a autorizácia, sledovanie a monitorovanie, a automatické opätovné skúšanie a odolnosť voči zlyhaniam.

**Istio** a **Linkerd** sú dva populárne nástroje, ktoré implementujú koncept service mesh. Oba poskytujú sadu funkcionalít na zlepšenie spoľahlivosti, bezpečnosti a observability mikroslužieb, pričom minimalizujú potrebu implementácie týchto funkcií priamo v mikroslužbách. Tým umožňujú vývojárom sústrediť sa viac na business logiku aplikácie namiesto riešenia infraštruktúrnych problémov.

**Istio**

* Istio je open-source service mesh, ktorý umožňuje ľahké definovanie pravidiel pre komunikáciu medzi službami, monitorovanie a sledovanie ich, ako aj zabezpečenie komunikácie. Istio pridáva tieto schopnosti na existujúce distribuované aplikácie bez potreby zmeny kódu týchto aplikácií, využitím tzv. "sidecar" kontajnerov, ktoré sú pridané do každej služby v Kubernetes clustri.

**Linkerd**

* Linkerd je ľahší a rýchlejší service mesh, zameraný na jednoduchosť a rýchlosť. Poskytuje základné funkcie service mesh ako sú load balancing, bezpečné komunikácie, sledovanie a observability. Linkerd je navrhnutý tak, aby bol čo najjednoduchšie nasaditeľný a použiteľný, s minimálnym vplyvom na výkon aplikácie.

Oba nástroje, Istio a Linkerd, sú príkladmi implementácie service mesh architektúry, ktoré zjednodušujú správu mikroslužieb tým, že premiestňujú komunikačné, bezpečnostné a monitorovacie funkcie do infraštruktúry, čím sa zníži zložitosť jednotlivých mikroslužieb.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Vieš čo je to event sourcing? používal si ho?**

Event Sourcing je architektonický vzor, ktorý sa používa na zachytávanie všetkých zmien stavu aplikácie ako postupnosť udalostí. Namiesto priameho ukladania súčasného stavu doménových objektov do databáze, Event Sourcing ukladá sériu udalostí, ktoré zaznamenávajú, ako sa stav objektu v priebehu času menil. Každá udalosť predstavuje zmenu stavu, a preto rekonštrukcia súčasného stavu objektu zahŕňa prehranie týchto udalostí od začiatku.

**Kľúčové vlastnosti Event Sourcing:**

* **Imutabilita udalostí**: Udalosti sú nemenné, čo znamená, že raz zaznamenané, nemôžu byť zmenené ani vymazané. Toto zachováva presný záznam o tom, čo sa stalo, a kedy to bolo.
* **Rekonštrukcia stavu**: Stav aplikácie alebo objektu môže byť kedykoľvek rekonštruovaný prehraním udalostí od začiatku do súčasnosti. To umožňuje ľahké vrátenie zmien alebo analyzovanie histórie stavu.
* **Auditovateľnosť a sledovateľnosť**: Ukladanie každej zmeny ako udalosti poskytuje kompletnú históriu všetkých zmien, čo zjednodušuje auditovanie a analýzu.
* **Oddelenie čítania a zápisu**: Event Sourcing často ide ruka v ruke s CQRS (Command Query Responsibility Segregation), kde sa operácie čítania a zápisu oddelia na zvýšenie výkonu, škálovateľnosti a bezpečnosti.

**Príklad použitia Event Sourcing:**

Predstavte si e-commerce systém, kde každá operácia s košíkom (pridanie položky, odstránenie položky, zmena množstva) generuje udalosť, ako je "ItemAddedToCart", "ItemRemovedFromCart" alebo "ItemQuantityChanged". Tieto udalosti sa ukladajú v chronologickom poradí. Ak chcete zistiť súčasný obsah košíka, systém prehrá všetky udalosti týkajúce sa tohto košíka, čím rekonštruuje jeho súčasný stav.

Event Sourcing je mocný vzor, ktorý prináša veľa výhod, najmä v systémoch vyžadujúcich vysokú úroveň auditovateľnosti alebo tam, kde je potrebné komplexné spracovanie a analýza histórie zmien stavu.

Jednoduchý príklad kódu v Jave, ktorý demonštruje základné princípy Event Sourcing. Predpokladajme, že máme jednoduchú aplikáciu na správu účtov, kde môžeme vkladať a vyberať peniaze. Každá operácia (vklad alebo výber) bude generovať udalosť, ktorá sa uloží do "event store" (udalostného úložiska). Následne môžeme stav účtu rekonštruovať prehraním týchto udalostí.

**Definícia základných udalostí**

Najprv definujeme základné rozhranie pre udalosti a konkrétne implementácie pre vklad a výber:

A screen shot of a computer program

Description automatically generated

**Implementácia účtu**

Potom definujeme jednoduchú triedu účtu, ktorá používa tieto udalosti na zmenu stavu:

A computer screen shot of a program

Description automatically generated

**Demonštrácia použitia**

A nakoniec, demonštrácia, ako by sa tieto komponenty mohli použiť na sledovanie operácií na účte a jeho rekonštrukciu:

**public class EventSourcingExample {**

**public static void main(String[] args) {**

**Account account = new Account();**

**account.deposit(100);**

**account.withdraw(50);**

**// Zobrazenie záznamu udalostí a súčasného stavu**

**account.printStatement();**

**// Rekonštrukcia účtu z udalostí**

**Account reconstructedAccount = Account.rebuildFromEvents(account.eventLog);**

**reconstructedAccount.printStatement(); // Malo by zobraziť rovnaký výstup ako pôvodný účet**

**}**

**}**

A screen shot of a computer code

Description automatically generated

Tento jednoduchý príklad ilustruje, ako Event Sourcing umožňuje sledovať zmeny stavu objektu prostredníctvom udalostí a ako je možné objekt rekonštruovať prehraním týchto udalostí. V reálnych aplikáciách by sa udalosti ukladali do trvalého úložiska a mohli by sa použiť na riešenie rôznych požiadaviek na auditovateľnosť, skalovateľnosť a komplexnú správu stavu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Vieš čo je concurrent hashmap a ako funguje?**

**ConcurrentHashMap** v Jave je trieda z balíka **java.util.concurrent**, ktorá poskytuje thread-safe implementáciu **Map** rozhrania. To znamená, že je navrhnutá tak, aby zabezpečila správnu funkčnosť a efektívnosť v prostredí s viacerými vláknami (threads) bez potreby synchronizácie celého mapového objektu.

**Kľúčové vlastnosti ConcurrentHashMap:**

* **Vysoká súbežnosť (concurrency)**: Na rozdiel od **Hashtable** alebo synchronizovaných máp (získaných cez **Collections.synchronizedMap()**), **ConcurrentHashMap** umožňuje viacerým vláknam súčasne čítať a zapisovať do mapy bez významného zníženia výkonu. Dosahuje to rozdelením mapy na rôzne segmenty, pričom každý segment môže byť zamykaný nezávisle.
* **Lock Stripping**: **ConcurrentHashMap** používa techniku známu ako lock stripping, kde zamknutie je aplikované len na segment alebo časť mapy namiesto celého mapového objektu. To umožňuje viacerým vláknam upravovať rôzne segmenty mapy súčasne, čo zvyšuje efektivitu pri práci s veľkými dátovými množstvami v súbežnom prostredí.
* **Neobsahuje žiadne zamknuté čítanie**: Čítanie hodnôt z **ConcurrentHashMap** nevyžaduje explicitné zamknutie, čo umožňuje vysoký počet súbežných čítacích operácií bez zablokovania. Zápis a aktualizácie však stále vyžadujú zamknutie na úrovni segmentu, aby sa zachovala konzistencia a thread-safety.
* **null hodnoty**: Na rozdiel od **Hashtable**, **ConcurrentHashMap** neumožňuje kľúče alebo hodnoty s hodnotou **null**. Pokus o vloženie **null** kľúča alebo hodnoty vyvolá **NullPointerException**.

**Ako funguje:**

Predstavte si **ConcurrentHashMap** ako mapu rozdelenú na N segmentov, kde každý segment je vlastne malá mapa, ktorá je chránená vlastným zámkom (lock). Keď vlákno chce vykonať operáciu (napríklad vloženie alebo odstránenie), najprv sa určí, do ktorého segmentu kľúč patrí (zvyčajne pomocou hashovacej funkcie). Potom vlákno získa zámok pre daný segment, vykoná potrebnú operáciu a následne zámok uvoľní. Týmto spôsobom môže viac vlákien pracovať s rôznymi segmentmi mapy súčasne, čím sa dosahuje vysoká súbežnosť bez globálneho zamknutia celého mapového objektu.

Tento prístup znižuje kontenčnosť zámkov (lock contention) - situáciu, kedy viaceré vlákna súťažia o získanie zámku na rovnakom objekte, čo vedie k výraznému zlepšeniu výkonu v súbežných aplikáciách.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Čo sú private resources, na co a kedy sa pouzivaju?**  
V kontexte softvérového inžinierstva a programovania, termín "private resources" (súkromné zdroje) môže mať rôzne významy v závislosti od kontextu, v ktorom je použitý. Však, základná idea sa týka zdrojov alebo dát, ktoré sú prístupné a používané výhradne v rámci určitého obmedzeného kontextu, ako je napríklad jedna aplikácia, modul, alebo dokonca určitá časť kódu. Tu sú niektoré príklady a vysvetlenia, ako a kedy sa používajú "private resources":

**1. Súkromné premenné v triedach**

* **Použitie**: V objektovo orientovanom programovaní, súkromné (private) premenné triedy sú prístupné len z metód tej istej triedy. Súkromné premenné sa používajú na zabránenie priamemu prístupu k dátam objektu zvonka triedy, čo zabezpečuje zapúzdrenie a ochranu stavu objektu.
* **Kedy sa používajú**: Na udržanie integrity objektu tým, že sa zabráni vonkajšiemu kódu v zmene vnútorných stavových premenných priamo, čím sa znižuje riziko chýb.

**2. Súkromné API**

* **Použitie**: Súkromné API sú rozhrania dostupné len pre určité komponenty alebo služby v rámci jednej aplikácie alebo medzi obmedzenou skupinou aplikácií. Nejde o verejné API, ktoré je dokumentované a určené na široké použitie.
* **Kedy sa používajú**: Na poskytnutie funkcionality, ktorá je špecifická pre interné procesy aplikácie alebo organizácie, a nie je určená na verejné použitie.

**3. Súkromné cloudové zdroje**

* **Použitie**: Súkromné cloudové prostredia alebo zdroje sú určené výhradne pre jednu organizáciu. Môžu byť hostované interným tímom alebo externým poskytovateľom, ale nie sú zdieľané s inými organizáciami.
* **Kedy sa používajú**: Keď organizácie potrebujú väčšiu kontrolu nad svojimi dátami a aplikáciami alebo musia spĺňať špecifické bezpečnostné alebo regulačné požiadavky.

**4. Súkromné repozitáre kódu**

* **Použitie**: Súkromné repozitáre umožňujú vývojárom uchovávať zdrojový kód aplikácií a projektov prístupný len pre autorizovaných používateľov alebo tímy.
* **Kedy sa používajú**: Na ochranu duševného vlastníctva, zabezpečenie projektov a zdrojového kódu pred neautorizovaným prístupom, alebo na správu prístupu k citlivým alebo konfidenčným projektom.

**Zhrnutie**

"Private resources" sú základným nástrojom v softvérovom inžinierstve na zabezpečenie bezpečnosti, ochrany duševného vlastníctva, a na udržiavanie čistoty a integrity kódu a dát. Ich použitie je kritické v mnohých aspektoch vývoja softvéru, od návrhu aplikácií až po správu infraštruktúry.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Closeable vs autocloseable? aký je rozdiel/vztah medzi nimi? ktoré dedí od ktorého?**

V Jave, rozhrania **Closeable** a **AutoCloseable** poskytujú mechanizmus pre zatváranie zdrojov, najmä v kontexte správy zdrojov ako sú súbory, databázové spojenia alebo sieťové spojenia. Hoci majú podobný účel, existujú medzi nimi určité rozdiely a majú rôzne použitia.

**AutoCloseable**

* Rozhranie **AutoCloseable** bolo zavedené v Jave 7 ako súčasť vylepšenia try-with-resources príkazu. Jeho hlavným účelom je poskytnúť obecný mechanizmus zatvárania zdrojov.
* Deklaruje metódu **void close() throws Exception**, ktorá môže vyvolať **Exception**. Toto je flexibilnejšie, pretože neobmedzuje vyvolané výnimky na konkrétny typ.
* Toto rozhranie je navrhnuté tak, aby bolo možné ho použiť so širokou škálou zdrojov, nielen s tými, ktoré pracujú so vstupom/výstupom (I/O).

**Closeable**

* Rozhranie **Closeable** je špeciálnejšie a bolo súčasťou Javy pred verziou 7. Je určené hlavne pre triedy, ktoré pracujú s I/O zdrojmi, ako sú prúdy (streams) a čítačky (readers).
* Deklaruje metódu **void close() throws IOException**, čo znamená, že pri zatváraní zdroja môže vyvolať iba **IOException**. Toto zúženie na **IOException** robí **Closeable** vhodnejším pre I/O operácie.
* Triedy pracujúce s I/O zdrojmi, ako sú **FileInputStream**, **FileOutputStream**, **RandomAccessFile**, implementujú **Closeable**, čo zjednodušuje správu týchto zdrojov v rámci try-with-resources príkazu.

**Vzťah medzi nimi**

* **Closeable** je podrozhraním **AutoCloseable**. To znamená, že všetko, čo implementuje **Closeable**, je automaticky považované za implementujúce **AutoCloseable**.
* Dôležitým rozdielom je, že zatiaľ čo **AutoCloseable** môže vyvolať akúkoľvek výnimku, **Closeable** je špecifické pre I/O operácie a obmedzuje vyvolané výnimky na **IOException**.

**Kedy použiť ktoré**

* **AutoCloseable**: Keď potrebujete všeobecný mechanizmus zatvárania pre zdroje, ktoré môžu vyvolať rôzne typy výnimiek, nie len **IOException**.
* **Closeable**: Pre triedy pracujúce s I/O, kde chcete obmedziť vyvolané výnimky na **IOException**.

Použitie týchto rozhraní v rámci try-with-resources príkazu zabezpečuje, že všetky zdroje budú riadne zatvorené na konci bloku, čím sa minimalizuje riziko únikov zdrojov.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Java stream api - aké 2 typy metód má?**

Java Stream API, ktoré bolo zavedené v Jave 8, poskytuje bohatý mechanizmus na prácu s kolekciami dát deklaratívnym spôsobom. Stream API má dva základné typy metód:

**1. Medzioperačné metódy (Intermediate Operations)**

Tieto metódy spracúvajú prvky streamu a vracajú stream ako výstup, čo umožňuje reťazenie viacerých operácií. Medzioperačné metódy sú "lazy", čo znamená, že nevykonávajú žiadne spracovanie, kým nie je vyvolaná terminálna operácia. Medzi základné medzioperačné metódy patria:

* **filter(Predicate<T>)**: Vracia stream, ktorý zahŕňa len tie prvky, ktoré vyhovujú danému predikátu.
* **map(Function<T, R>)**: Aplikuje funkciu na prvky streamu a vracia stream s výsledkami aplikácie funkcie.
* **flatMap(Function<T, Stream<R>>)** : "Sploští" viacero streamov do jedného streamu.
* **distinct()**: Vracia stream, ktorý obsahuje len unikátne prvky podľa metódy **equals(Object)**.
* **sorted()**: Vracia stream, ktorý je zoradený podľa prírodného usporiadania alebo podľa poskytnutého komparátora.
* **peek(Consumer<T>)**: Vykonáva zadanú akciu na každom prvku streamu ako prostriedok na debugovanie.

**2. Terminálne metódy (Terminal Operations)**

Terminálne metódy spustia spracovanie prvkov streamu a vracajú výsledok alebo vykonajú vedľajší efekt. Po vyvolaní terminálnej operácie sa stream stáva spotrebovaným a už nie je možné ho používať pre ďalšie operácie. Medzi terminálne operácie patria:

* **forEach(Consumer<T>)**: Vykoná zadanú akciu pre každý prvok streamu.
* **toArray()**: Vracia pole, ktoré obsahuje všetky prvky streamu.
* **reduce(BinaryOperator<T>)**: Používa redukčnú operáciu na prvky streamu a vracia voliteľný výsledok.
* **collect(Collector<T, A, R>)**: Prenesie prvky do novej formy, napríklad do zoznamu, množiny alebo mapy, použitím rôznych stratégií definovaných v triede **Collectors**.
* **anyMatch(Predicate<T>)**, **allMatch(Predicate<T>)**, **noneMatch(Predicate<T>)**: Vracajú boolean hodnotu po vyhodnotení, či nejaký, všetky, alebo žiadne prvky streamu vyhovujú zadanému predikátu.
* **findFirst()**, **findAny()**: Vracajú **Optional** opisujúce nájdený prvok.

Každý stream musí obsahovať konfiguráciu medzioperačných metód nasledovanú jednou terminálnou operáciou, ktorá spustí proces spracovania dát.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**Intermediate vs terminated operations?**

V kontexte Java Stream API, rozdiel medzi medzioperačnými (Intermediate) a terminálnymi (Terminal) operáciami spočíva v ich funkcii, spôsobe vykonávania a výsledkoch, ktoré produkujú. Tieto dva typy operácií spoločne umožňujú efektívne spracovanie kolekcií dát.

**Medzioperačné Operácie (Intermediate Operations)**

* **Lenivosť (Laziness)**: Sú "lazy", čo znamená, že sa nevykonajú okamžite. Skutočné spracovanie dát sa začne až pri vyvolaní terminálnej operácie.
* **Typ výstupu**: Vracajú nový stream ako svoj výsledok, čo umožňuje reťazenie viacerých medzioperačných operácií. Tento prístup podporuje funkcionálnu kompozíciu a prehľadnosť kódu.
* **Stavová vs. Bezstavová**: Niektoré medzioperačné operácie môžu byť stavové (napr. **distinct()**, **sorted()**), čo znamená, že môžu potrebovať uchovať stav počas spracovania dát, zatiaľ čo bezstavové operácie (napr. **map()**, **filter()**) spracúvajú každý prvok nezávisle na ostatných.
* **Príklady**: **map()**, **filter()**, **flatMap()**, **peek()**, **distinct()**, **sorted()**.

**Terminálne Operácie (Terminal Operations)**

* **Spúšťací mechanizmus**: Sú vykonávané okamžite a spúšťajú spracovanie všetkých operácií v reťazci streamu, vrátane medzioperačných operácií. Po vykonaní terminálnej operácie sa stream stáva "spotrebovaným" a nemožno ho ďalej používať.
* **Typ výstupu**: Výstupom môže byť hodnota (alebo objekt), ako je suma alebo zoznam, alebo môže vykonať vedľajší efekt, ako je výpis prvkov. Terminálne operácie teda nevracajú stream.
* **Príklady**: **forEach()**, **collect()**, **reduce()**, **anyMatch()**, **allMatch()**, **noneMatch()**, **findFirst()**, **findAny()**, **toArray()**.

**Kľúčové Rozdiely**

* **Vykonávanie**: Medzioperačné operácie sa nevykonávajú ihneď a sú "lazy", zatiaľ čo terminálne operácie spúšťajú vykonávanie celého reťazca operácií na streame.
* **Výstup**: Medzioperačné operácie vracajú stream, umožňujúc tým reťazenie ďalších operácií. Terminálne operácie vracajú konkrétny typ výstupu alebo vykonávajú akciu, a tým končia reťaz operácií.
* **Použitie streamu**: Po vykonaní terminálnej operácie je stream spotrebovaný a nemožno ho ďalej používať, zatiaľ čo medzioperačné operácie tento stav nespôsobujú.

Pochopenie týchto rozdielov je kľúčové pre efektívne využívanie Stream API v Jave, umožňuje to vývojárom písať čistý, modulárny a efektívny kód pre spracovanie kolekcií dát.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Typy/implementacie lambda functions?**

Lambda funkcie, známe aj ako anonymné funkcie, sú výkonným nástrojom v mnohých programovacích jazykoch vrátane Javy. Od Javy 8, ktorá bola vydaná v marci 2014, Java podporuje lambda výrazy, čo umožňuje písať kompaktný a efektívny kód pre funkčné rozhrania.

V Jave sa lambda výrazy zvyčajne používajú v kontexte funkčných rozhraní. Funkčné rozhranie je rozhranie, ktoré obsahuje práve jeden abstraktný metód. Java API poskytuje množstvo štandardných funkčných rozhraní v balíčku **java.util.function**, ako sú **Predicate<T>**, **Function<T,R>**, **Consumer<T>** a **Supplier<T>**.

**Syntax lambda výrazu**

Základná syntax lambda výrazu v Jave je:



Kde:

* **parametre** môžu byť jeden parameter bez typu (ak je z kontextu jasné, o aký typ ide) alebo viac parametrov v zátvorkách, s uvedením typu alebo bez neho.
* **telo** môže byť jednoduchý výraz alebo blok kódu ohraničený zloženými zátvorkami **{}**.

**Príklady Lambda výrazov**

A screenshot of a computer

Description automatically generated

**Implementácia pomocou Lambda výrazov**

Lambda výrazy sa dajú použiť na implementáciu metódy definovanej funkčným rozhraním. Toto umožňuje, napríklad, efektívne implementovať poslucháčov udalostí, pracovať s kolekciami pomocou stream API a vykonávať iné funkčné operácie bez potreby vytvárať anonymné triedy.

**Zhrnutie**

Lambda výrazy v Jave umožňujú písať kompaktný, funkčný kód. Vďaka nim je práca s kolekciami, udalosťami a API pre funkčné programovanie efektívnejšia a prehľadnejšia. Je dôležité si však uvedomiť, že lambda výrazy sú možné len v kontexte funkčných rozhraní.Top of Form

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ktoré inštancie vieš zapísať ako lambdu?**

V Jave možno lambda výrazy použiť na akékoľvek funkčné rozhranie, čo je rozhranie s presne jedným abstraktným metódom. Tu sú niektoré bežné príklady funkčných rozhraní dostupných v Java API, ktoré možno implementovať pomocou lambda výrazov:

A screenshot of a computer program

Description automatically generated

Tieto príklady ukazujú, ako možno rôzne typy funkčných rozhraní implementovať pomocou lambda výrazov, čím sa výrazne zjednodušuje a zefektívňuje písanie kódu v Jave. Lambda výrazy umožňujú vyjadriť inštancie funkčných rozhraní v kompaktnej forme, čo je obzvlášť užitočné pri práci s kolekciami, streamami a pri implementácii poslucháčov udalostí.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Vieš na čo slúži offset v Kafke?**  
Offset v Apache Kafka je kľúčový koncept, ktorý umožňuje spotrebiteľom (consumers) sledovať, ktoré správy už boli prečítané zo zoznamu (topic). Kafka ukladá správy v partíciách v rámci tém, pričom každá správa v partícii má jedinečné sériové číslo známe ako "offset".

**Funkcia Offsetu**

Offset slúži niekoľkým dôležitým účelom v Kafke:

1. **Sledovanie Postupu Spotrebitelia:**
   * Každý spotrebiteľ alebo skupina spotrebiteľov udržuje záznam o offsete správ, ktoré už boli prečítané, čo im umožňuje pokračovať v čítaní správ z miesta, kde skončili. Toto je kriticky dôležité pre zabezpečenie, že spotrebiteľ nepremešká žiadne správy a zároveň nečíta správy viackrát.
2. **Podpora Replikácie:**
   * Offsety umožňujú Kafke efektívne replikovať dáta medzi uzlami a zabezpečiť tak vysokú dostupnosť a odolnosť voči chybám.
3. **Návratovosť a Preskúmanie:**
   * Spotrebitelia môžu nastaviť svoje offsety na staršie hodnoty, ak potrebujú znovu spracovať správy z určitého bodu. To umožňuje aplikáciám obnoviť spracovanie od konkrétneho bodu v prípade chyby alebo keď sú zavedené nové spracovateľské logiky.
4. **Paralelné Spracovanie:**
   * Vďaka offsetom môžu byť správy v partícii spracované paralelne rôznymi spotrebiteľmi v rámci spotrebiteľskej skupiny, pričom každý spotrebiteľ vie, ktorú časť dát má spracovať.

**Ako Offsety Fungujú**

* **Ukladanie Offsetov:** Spotrebiteľské skupiny ukladajú svoje offsety buď lokálne alebo v špeciálnej téme Kafka nazvanej **\_\_consumer\_offsets**, čo umožňuje spotrebiteľom pokračovať presne tam, kde skončili, aj po reštartovaní alebo zlyhaní.
* **Automatické Spravovanie Offsetov:** Kafka poskytuje konfigurácie, ktoré umožňujú automatické potvrdzovanie offsetov po úspešnom spracovaní správ. Spotrebitelia môžu tiež manuálne spravovať offsety pre pokročilé spracovateľské modely.

Offsety sú teda základnou súčasťou toho, ako Kafka poskytuje spoľahlivé, škálovateľné a vysoko dostupné správy streamovacie služby. Pomáhajú zabezpečiť, že správy sú spracované správne a umožňujú komplexné spracovateľské vzory, ako je preskúmanie správ a paralelné spracovanie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Čo je immutable object? Načo a kedy ho využívaš?**  
Immutable objekt je objekt, ktorého stav sa nedá zmeniť po jeho vytvorení. To znamená, že akonáhle je objekt vytvorený s určitými hodnotami, tieto hodnoty zostanú počas celého životného cyklu objektu nezmenené.

**Načo a kedy využívať immutable objekty?**

1. **Zvýšenie bezpečnosti:** Keďže stav immutable objektu sa nedá zmeniť, sú tieto objekty prirodzene bezpečné voči zmene a môžu byť bezpečne zdieľané medzi rôznymi časťami aplikácie bez rizika nechcených zmien.
2. **Zjednodušenie paralelného programovania:** V prostredí s viacerými vláknami sú zdieľané mutable objekty zdrojom zložitých problémov súvisiacich so súčasnosťou, ako sú preteky o zdroje (race conditions) a blokovania. Immutable objekty eliminujú tieto problémy, pretože ich stav sa nemôže zmeniť, takže sú prirodzene bezpečné pre použitie vo viacvláknových aplikáciách.
3. **Optimalizácia výkonu:** Niektoré systémy môžu využívať nemeniteľnosť na optimalizáciu výkonu, napríklad cachovanie objektov, pretože immutable objekty môžu byť bezpečne znovu použité alebo zdieľané.
4. **Podpora funkcionálneho programovania:** Immutable objekty sú základným stavebným kameňom funkcionálneho programovania, pretože podporujú koncepty ako čisté funkcie a nemenné dáta, čo vedie k predvídateľnejším a jednoduchším na pochopenie programom.

**Kedy ich využiť**

* **Bezpečné zdieľanie dát medzi vláknami:** V aplikáciách, kde viacero vlákien pristupuje a manipuluje s rovnakými dátami, immutable objekty zabezpečujú, že dáta zostanú konzistentné a nebudú nesprávne modifikované.
* **Vytvorenie konštantných hodnôt:** Ak potrebujete reprezentovať hodnoty, ktoré sa nemajú meniť (napríklad konfiguračné nastavenia, globálne konštanty).
* **Funkcionálne alebo reaktívne programovacie modely:** V týchto paradigmatách je nemennosť kľúčová pre zabezpečenie predvídateľného toku dát a vedie k čistejšiemu a jednoduchšiemu kódu.

V Jave sa často využívajú immutable objekty, najmä v kolekciách z Java Collections Framework, ako sú **String**, **BigDecimal**, a všetky obaly primitívnych typov (**Integer**, **Long** atď.). Vytvorenie vlastných immutable objektov vyžaduje, aby všetky polia boli označené ako **final** a aby objekt neumožňoval zmeny svojho stavu po inicializácii, či už priamo alebo nepriamo cez referencie na mutable objekty.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi mutable va immutable objects?**

Rozdiel medzi mutable (meniteľnými) a immutable (nemeniteľnými) objektami spočíva v možnosti zmeny ich stavu po vytvorení.

**Mutable Objekty**

* **Zmeniteľnosť:** Mutable objekty môžu meniť svoj stav alebo dáta, ktoré obsahujú, po ich vytvorení. To znamená, že ich metódy môžu meniť ich vlastnosti alebo stav bez potreby vytvorenia novej inštancie objektu.
* **Použitie:** Sú vhodné pre aplikácie, kde je potrebné meniť stav objektu, ako sú dynamické zoznamy, zásobníky, fronty a iné štruktúry dát, ktoré sa často menia.
* **Príklady:** V Jave sú príkladmi mutable objektov triedy ako **StringBuilder**, **ArrayList**, **HashMap** a iné kolekcie.

**Immutable Objekty**

* **Nemeniteľnosť:** Immutable objekty, na druhej strane, nemôžu meniť svoj stav ani dáta, ktoré obsahujú, po ich vytvorení. Akákoľvek "zmena" v takomto objekte v skutočnosti vyžaduje vytvorenie novej inštancie objektu s požadovaným stavom.
* **Použitie:** Sú ideálne pre zdieľanie bezpečného prístupu medzi viacerými vláknami, pretože ich stav sa nemôže zmeniť, čo eliminuje problémy súvisiace so súčasnosťou. Sú tiež vhodné pre objekty, ktoré by mali reprezentovať pevné hodnoty, ako sú konfiguračné nastavenia alebo pevné referenčné hodnoty.
* **Príklady:** Typickými príkladmi sú **String**, **Integer** a **BigDecimal** v Jave.

**Kľúčové Rozdiely**

1. **Bezpečnosť a Súčasnosť:** Immutable objekty sú bezpečnejšie v prostrediach s viacerými vláknami, pretože ich nemôže žiadne vlákno zmeniť, čím sa predchádza problémom súčasného prístupu.
2. **Výkon:** Mutable objekty môžu byť v niektorých prípadoch efektívnejšie, pokiaľ ide o pamäť a čas, pretože umožňujú zmeny bez potreby vytvárať nové objekty. Na druhej strane, immutable objekty môžu zlepšiť výkon v prostredí s viacerými vláknami tým, že eliminujú potrebu synchronizácie.
3. **Návrh a Údržba Kódu:** Kód s immutable objektmi je často jednoduchší na pochopenie a udržiavanie, keďže tok dát je predvídateľnejší a neexistujú žiadne vedľajšie efekty menení objektov. Mutable objekty môžu zvýšiť zložitosť, najmä v veľkých a komplexných systémoch.

Voľba medzi použitím mutable alebo immutable objektov závisí od konkrétnych potrieb aplikácie, ako aj od preferovaných vlastností, ako sú bezpečnosť, efektívnosť a jednoduchosť kódu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké cache má Hibernate?**

Hibernate poskytuje dva hlavné typy cache na optimalizáciu výkonu aplikácií pracujúcich s databázou:

**1. First-Level Cache (Cache prvej úrovne)**

* **Scope:** Je automaticky povolená a je asociovaná s **Session** objektom Hibernate. Každá session má svoju vlastnú first-level cache.
* **Účel:** Minimalizuje počet dotazov do databáze tým, že ukladá entity, ktoré boli načítané v rámci aktuálnej session. Ak sa entita požaduje viackrát v rámci tej istej session, Hibernate ju vráti z cache prvej úrovne namiesto toho, aby vykonal ďalší dotaz do databáze.
* **Životnosť:** Existuje len počas trvania session. Keď sa session zatvorí, cache sa vyčistí.

**2. Second-Level Cache (Cache druhej úrovne)**

* **Scope:** Je voliteľná a konfigurovateľná. Na rozdiel od cache prvej úrovne, cache druhej úrovne je globálna pre celú aplikáciu a môže byť zdieľaná medzi viacerými sessions.
* **Účel:** Umožňuje ukladanie dát medzi transakciami a sessions. Môže sa používať na cachovanie entity, kolekcie, dotazov a dokonca aj prírodných dotazov SQL. Táto cache znižuje potrebu vykonávania častých dotazov do databáze tým, že udržuje dáta dlhodobo dostupné medzi viacerými requests.
* **Konfigurácia:** Vyžaduje explicitnú konfiguráciu. Musíte určiť, ktoré entity alebo kolekcie by mali byť cachované, a zvoliť implementáciu cache druhej úrovne, ako je napríklad Ehcache, Infinispan alebo Hazelcast.

**Query Cache (Cache dotazov)**

* **Účel:** Hibernate tiež podporuje cachovanie výsledkov dotazov. Táto cache ukladá výsledky dotazov spolu s identifikátormi entít, na ktoré sa dotaz vzťahuje. Je užitočná pre cachovanie výsledkov často používaných dotazov.
* **Použitie:** Aby bolo možné použiť query cache, musí byť povolená second-level cache. Query cache vyžaduje dodatočné zdroje a môže byť zložité ju správne nastaviť, preto je dôležité ju používať uvážlivo.

**Význam cache v Hibernate**

Caching v Hibernate je kľúčovou funkciou pre zlepšenie výkonu aplikácií tým, že zníži záťaž na databázu a zrýchli prístup k dátam. Efektívne využitie cache môže výrazne znížiť latenciu a zvýšiť rýchlosť odpovede aplikácie. Je však dôležité správne konfigurovať a spravovať cache, aby sa zabránilo nekonzistencii dát a zabezpečila aktualizácia cache v prípade zmeny dát v databáze.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je scope beanov v Springu?**  
Spring Framework poskytuje rôzne scope (rozsahy) pre beany, ktoré určujú, ako sú inštancie beanov spravované a ako dlho existujú. Tieto scope umožňujú vývojárom prispôsobiť životný cyklus a viditeľnosť beanov v aplikácii podľa ich potrieb. Nižšie sú uvedené základné scope, ktoré Spring poskytuje:

**1. Singleton**

* **Popis:** Predvolený scope. Spring vytvorí a spravuje práve jednu inštanciu beanu na Spring IoC kontajner. Všetky požiadavky na bean s týmto scope budú vrátené s tou istou inštanciou.
* **Použitie:** Vhodné pre beany, ktoré sú bezstavové alebo keď je potrebná iba jedna inštancia pre celú aplikáciu.

**2. Prototype**

* **Popis:** Pre každú požiadavku na bean vytvorí Spring novú inštanciu. To znamená, že každé **getBean()** volanie alebo injektáž závislostí pre tento typ beanu vytvorí nový objekt.
* **Použitie:** Vhodné pre beany, ktoré sú určené pre krátkodobé použitie alebo ktoré udržiavajú stav, ktorý nie je zdieľaný medzi rôznymi časťami aplikácie.

**3. Request**

* **Popis:** Bean je vytvorený a existuje počas jednej HTTP požiadavky. To znamená, že pre každú novú HTTP požiadavku je vytvorená nová inštancia beanu.
* **Použitie:** Vhodné pre beany, ktoré udržiavajú informácie špecifické pre HTTP požiadavku.

**4. Session**

* **Popis:** Bean je vytvorený a existuje počas jednej HTTP session. Každá session má svoju vlastnú inštanciu beanu.
* **Použitie:** Vhodné pre beany, ktoré udržiavajú informácie špecifické pre session, ako sú napríklad používateľské preferencie.

**5. Application**

* **Popis:** Bean je vytvorený a existuje na úrovni ServletContext. To znamená, že bean je zdieľaný medzi všetkými sessions v aplikácii.
* **Použitie:** Vhodné pre beany, ktoré udržiavajú globálne stavové informácie dostupné pre celú aplikáciu.

**6. WebSocket**

* **Popis:** Bean je vytvorený a existuje v rámci životného cyklu WebSocket spojenia.
* **Použitie:** Vhodné pre beany, ktoré sú špecifické pre WebSocket komunikáciu.

Výber správneho scope pre bean je dôležitý, pretože ovplyvňuje nielen správu pamäti, ale aj to, ako sú dáta zdieľané a izolované medzi rôznymi časťami aplikácie. Vývojári by mali zvážiť požiadavky svojej aplikácie a správne vybrať scope pre každý bean, aby optimalizovali výkon a zabezpečili správnu funkčnosť.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi Hibernate a JPA**  
Hibernate a Java Persistence API (JPA) sú dva kľúčové pojmy v svete Java aplikácií pre prácu s databázami, ale majú rôzne úlohy a charakteristiky. Tu je prehľad hlavných rozdielov medzi nimi:

**Hibernate**

* **Implementácia:** Hibernate je jeden z najstarších a najpopulárnejších ORM (Object-Relational Mapping) nástrojov v Jave. Umožňuje mapovanie a správu databázových entít na Java objekty a naopak, čím zjednodušuje databázové operácie.
* **Funkcie:** Poskytuje bohatý súbor funkcií nad rámec štandardných JPA, vrátane dedičnosti, lazy loadingu, cachingu, HQL (Hibernate Query Language) a mnoho ďalších.
* **Nestandardné riešenia:** Môže obsahovať niektoré špecifické funkcie, ktoré nie sú súčasťou JPA špecifikácie, a preto môžu byť závislé na tejto konkrétnej implementácii.

**Java Persistence API (JPA)**

* **Špecifikácia:** JPA je štandardizované API definované v rámci Java EE platformy, ktoré poskytuje špecifikáciu pre ORM nástroje v Jave. Cieľom je umožniť vývojárom prácu s relačnými dátami na vysokej úrovni abstrakcie.
* **Implementácia:** JPA samo osebe nie je implementáciou; je to sada rozhraní a anotácií, ktoré definujú, ako by ORM nástroj mal pracovať. Hibernate je jednou z mnohých implementácií JPA (iné známe implementácie zahŕňajú EclipseLink, OpenJPA atď.).
* **Portabilita:** Keďže JPA je štandardizovaná, aplikácie napísané s využitím JPA sú teoreticky prenositeľné medzi rôznymi implementáciami bez potreby zmeny kódu.

**Kľúčové rozdiely**

1. **Úroveň abstrakcie:** Hibernate je konkrétna implementácia, zatiaľ čo JPA je iba špecifikácia, ktorej cieľom je umožniť jednotný prístup k ORM riešeniam.
2. **Funkčnosť:** Hibernate poskytuje viac funkcií nad rámec štandardného JPA, vrátane niektorých pokročilých optimalizácií a techník správy relačných dát.
3. **Portabilita:** Kód napísaný s použitím JPA je v teórii prenositeľný medzi rôznymi ORM implementáciami, zatiaľ čo kód špecifický pre Hibernate môže byť závislý na tejto platforme.

Vo výbere medzi použitím čistého JPA alebo Hibernate (alebo inej implementácie JPA) by vývojári mali zvážiť svoje konkrétne potreby vrátane požadovaných funkcií, požiadaviek na portabilitu a osobných preferencií vzhľadom na dostupnosť a zrelosť implementácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi Spring Data JPA a JDBC?**  
Spring Data JPA a JDBC sú dve rôzne technológie používané na prístup k dátam v aplikáciách Java, ale slúžia na rôzne účely a pracujú na rozličných úrovniach abstrakcie. Tu je prehľad hlavných rozdielov medzi nimi:

**JDBC (Java Database Connectivity)**

* **Úroveň abstrakcie:** JDBC je nízkoúrovňové API pre interakciu s databázami v Jave. Umožňuje priamu prácu s SQL príkazmi a spravuje pripojenie k databáze, vykonávanie SQL príkazov a spracovanie výsledkov.
* **Flexibilita:** Poskytuje vysokú úroveň kontroly nad databázovými operáciami, ale zároveň vyžaduje, aby vývojári manuálne spravovali pripojenia k databáze, SQL príkazy a výsledkové sady.
* **Práca s dátami:** Vývojári musia písať specifický SQL kód pre každú databázu, čo môže viesť k väčšiemu množstvu kódu a potenciálnym chybám.

**Spring Data JPA**

* **Úroveň abstrakcie:** Spring Data JPA je vysokoúrovňový modul/framework, ktorý stojí nad JPA a poskytuje ďalšiu vrstvu abstrakcie pre správu relačných dát. Uľahčuje prístup k dátam a ich manipuláciu prostredníctvom objektovo-relačného mapovania (ORM).
* **Simplifikácia:** Zjednodušuje vývoj dátových prístupových vrstiev tým, že poskytuje konvencie a generuje kód za bežných situácií, čo znižuje potrebu písania boilerplate kódu.
* **Práca s dátami:** Umožňuje definovať prístup k dátam pomocou rozhraní, ktoré Spring automaticky implementuje, čo minimalizuje potrebu priameho písania SQL. Podporuje aj koncepty ako CRUD operácie (Create, Read, Update, Delete), stránkovanie a triedenie, a to všetko s minimálnym alebo bez explicitného SQL kódu.

**Kľúčové rozdiely**

1. **Úroveň abstrakcie:** JDBC je nízkoúrovňové API, ktoré vyžaduje explicitné riadenie databázových operácií, zatiaľ čo Spring Data JPA poskytuje vysokoúrovňovú abstrakciu nad JPA, čo umožňuje efektívnejšiu prácu s dátami a zjednodušuje vývoj.
2. **Produktivita vývojárov:** Spring Data JPA zvyšuje produktivitu tým, že redukuje množstvo potrebného kódu na implementáciu bežných dátových prístupových operácií, zatiaľ čo JDBC vyžaduje od vývojárov, aby manuálne spravovali mnohé aspekty práce s databázou.
3. **Prístup k dátam:** Spring Data JPA abstrahuje potrebu priameho písania SQL príkazov prostredníctvom použitia repository rozhraní a ORM, zatiaľ čo JDBC si vyžaduje explicitné písanie a správu SQL príkazov.

Voľba medzi použitím Spring Data JPA alebo JDBC závisí od konkrétnych potrieb aplikácie, úrovne kontroly, ktorú vývojári potrebujú nad databázovými operáciami, a preferencie vyššej produktivity a jednoduchosti na úkor nižšej úrovne abstrakcie a kontrol.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Connection pooling - na čo slúži?**

Connection pooling je technika používaná v aplikáciách, ktoré interagujú s databázami, na zlepšenie výkonu a efektívnosti pri spracovaní databázových požiadaviek. Hlavným cieľom connection pooling je znížiť náklady spojené s otváraním a zatváraním databázových spojení, ktoré sú časovo náročné a zdrojovo nákladné operácie. Tu je niekoľko kľúčových bodov, ktoré vysvetľujú, ako connection pooling funguje a na čo slúži:

1. **Predchádzanie opakovanej inicializácii:** Bez connection poolu musí aplikácia otvoriť nové spojenie s databázou pri každej požiadavke na prístup k dátam, čo zahŕňa overenie pravosti, alokáciu zdrojov, a potom zatvorenie a uvoľnenie zdrojov po dokončení operácie. Tento proces je náročný na zdroje a čas.
2. **Zníženie latencie:** Connection pooling umožňuje aplikáciám rýchlo získať prístup k existujúcemu, už otvorenému spojeniu z poolu (skupiny preddefinovaných a otvorených spojení), čím sa výrazne znižuje latencia spojená s otváraním nových spojení.
3. **Efektívne využitie zdrojov:** Pooling umožňuje efektívnejšie využívanie databázových spojení tým, že zdieľa limitovaný počet spojení medzi viacerými používateľmi alebo procesmi. To znamená, že aplikácia môže obsluhovať viac požiadaviek s menej spojeniami, čo vedie k nižšej záťaži na databázový server.
4. **Zlepšená škálovateľnosť:** Aplikácie, ktoré využívajú connection pooling, sú lepšie škálovateľné, pretože môžu efektívnejšie spracovať zvýšený počet požiadaviek bez potreby lineárneho zvyšovania počtu databázových spojení.
5. **Konfigurovateľnosť a správa:** Connection pooly sú konfigurovateľné, čo umožňuje vývojárom nastaviť rôzne parametre, ako sú maximálny počet spojení v poole, čas života spojenia, časový limit na získanie spojenia z poolu a ďalšie. To pomáha optimalizovať výkon a zabezpečiť stabilnú a efektívnu správu zdrojov.

Connection pooling je široko využívaný v moderných aplikáciách, kde efektívny prístup k databáze je kritický pre celkový výkon a používateľskú skúsenosť. Technológie ako JDBC (Java Database Connectivity) v Jave, SQLAlchemy v Python-e a mnohé ďalšie ORM (Object-Relational Mapping) nástroje poskytujú vstavanú podporu pre connection pooling.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Poznáš/používal si Outbox pattern?**

Outbox pattern je návrhový vzor používaný v softvérovom inžinierstve, najmä v aplikáciách zameraných na mikroslužby alebo distribuované systémy, na zabezpečenie spoľahlivej komunikácie medzi službami a zachovanie konzistentnosti dát medzi rôznymi službami alebo databázami. Hoci ako AI nemám osobné skúsenosti s používaním technológií, môžem poskytnúť prehľad o tom, ako Outbox pattern funguje a kde sa využíva.

**Ako Outbox pattern funguje:**

1. **Zápis do Outboxu:** Keď aplikácia vykonáva operáciu, ktorá vyžaduje aktualizáciu dát a zároveň odoslanie správy do inej služby (napr. udalosť o zmene stavu), namiesto priameho odoslania správy do cieľovej služby alebo použitia udalostnej správy sa záznam najprv zapíše do špeciálnej tabuľky v databáze aplikácie nazvanej "outbox".
2. **Relayer alebo Event Publisher:** Samostatný proces, ktorý beží v pozadí (napríklad služba, ktorá pravidelne kontroluje tabuľku outboxu), potom skontroluje tabuľku outboxu, identifikuje nové záznamy a odosiela príslušné správy do cieľových služieb alebo do systému na spracovanie udalostí (napr. Kafka).
3. **Zmazanie alebo označenie spracovaných záznamov:** Po úspešnom odoslaní správy sa záznam v outboxe buď odstráni alebo označí ako spracovaný, čím sa zabezpečí, že správa nebude odoslaná viackrát.

**Výhody Outbox pattern:**

* **Atómovosť:** Tento vzor umožňuje atómové vykonanie databázových operácií a odosielania správ, čím sa zabráni problémom spojeným s dvojitým zápisom alebo stratenými správami.
* **Spoľahlivosť:** V prípade zlyhania aplikácie alebo problémov siete zabezpečuje, že žiadna správa nezanikne, pretože správy sú uložené v databáze a môžu byť odoslané po obnovení chyby.
* **Konzistencia:** Pomáha zachovať konzistenciu dát medzi mikroslužbami v distribuovaných systémoch tým, že zabezpečuje, že všetky zmeny stavu sú správne propagované.

**Použitie:**

Outbox pattern je obzvlášť užitočný v systémoch, kde je dôležitá vysoká spoľahlivosť správ, konzistencia dát naprieč službami alebo tam, kde sú potrebné komplexné transakcie zahŕňajúce viacero služieb. Je to dôležitá súčasť implementácie vzorov ako Event Sourcing alebo CQRS (Command Query Responsibility Segregation) a je široko používaný v moderných mikroslužbových architektúrach.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Čo Ti hovorí JAVA Mesh?**

Termín "Java Mesh" mi nie je priamo známy a neodkazuje na široko rozpoznaný koncept alebo technológiu v oblasti softvérového inžinierstva alebo v kontexte programovacieho jazyka Java, aspoň do doby môjho posledného update v apríli 2023. Môže ísť o špecifický produkt, knižnicu, framework, alebo koncept, ktorý sa objavil alebo získal popularitu po tomto dátume, alebo je možné, že je to termín používaný v špecifickom kontexte alebo projektu, s ktorým nie som oboznámený.

V oblasti softvérového inžinierstva sa termín "mesh" často používa v kontexte "service mesh" v architektúrach založených na mikroslužbách. Service mesh je infraštruktúrna vrstva zameraná na správu komunikácie medzi službami, zabezpečenie, monitorovanie a iné cezhraničné obavy. Populárne implementácie service mesh zahŕňajú Istio, Linkerd a Consul, ktoré sú technológie nezávislé od programovacieho jazyka a môžu byť použité v aplikáciách napísaných v Jave.

Ak máte na mysli niečo konkrétne súvisiace s "Java Mesh", mohlo by to byť niečo nové alebo menej známe, a odporúčam konzultovať špecifickú dokumentáciu alebo zdroje, ktoré sú priamo spojené s týmto termínom.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi WAIT a NOTIFY (v OBJECT)**

Metódy **wait()** a **notify()** (a **notifyAll()**) v Jave sú základné prostriedky pre komunikáciu medzi vláknami, ktoré umožňujú vláknam čakať na určitú podmienku a byť o jej zmene informované. Tieto metódy sú definované v triede **Object**, čo znamená, že každý objekt v Jave môže byť použitý ako zámok (monitor) pre koordináciu medzi vláknami. Tu je prehľad rozdielov a použitia **wait()** a **notify()**:

**wait()**

* **Použitie:** Metóda **wait()** je používaná v synchronizovanom bloku alebo metóde, aby aktuálne vlákno uvoľnilo zámok a čakalo, až kým iné vlákno nevyvolá **notify()** alebo **notifyAll()** na tom istom objekte (zámku). Toto uvoľnenie zámku umožňuje iným vláknam získať zámok a vykonať nejakú operáciu.
* **Efekt:** Keď vlákno volá **wait()**, prechádza do stavu čakania a uvoľňuje zámok, ktorý držalo. Zostane v tomto stave, až kým nie je iným vláknom vyvolané **notify()** alebo **notifyAll()** na tom istom objekte.

**notify()**

* **Použitie:** Metóda **notify()** tiež vyžaduje, aby bola volaná v synchronizovanom bloku alebo metóde. Slúži na oznámenie jednému z vlákien, ktoré čakajú na danom objekte (zámku), že môžu pokračovať. Výber vlákna, ktoré bude oznámené, je na úrovni JVM a nemôže byť špecifikovaný.
* **Efekt:** **notify()** prebudí jedno z čakajúcich vlákien. Toto vlákno potom bude môcť pokračovať po získaní zámku objektu.

**notifyAll()**

* Je to podobné **notify()**, ale namiesto toho, aby prebudilo len jedno vlákno, **notifyAll()** prebudí všetky vlákna, ktoré čakajú na danom objekte. Toto je užitočné, keď viac vlákien čaká na rôzne podmienky, ktoré môžu byť splnené rovnakou operáciou.

**Kľúčové Rozdiely**

* **Počet prebudených vlákien:** **wait()** spôsobí, že vlákno prejde do stavu čakania. **notify()** prebudí práve jedno z týchto čakajúcich vlákien, zatiaľ čo **notifyAll()** prebudí všetky čakajúce vlákna.
* **Účel:** **wait()** je používaná na čakanie na nejakú podmienku. **notify()** a **notifyAll()** sú používané na signalizáciu ostatným vláknam, že podmienka, na ktorú možno čakali, sa mohla zmeniť.

Pri používaní týchto metód je dôležité pamätať na to, že musia byť volané z synchronizovaného kontextu - to znamená, že vlákno musí mať zámok na objekte, na ktorom sú tieto metódy volané. To zabezpečuje, že manipulácia so stavom, ktorý je spoločný pre viaceré vlákna, je bezpečná a nedochádza k súbehovým podmienkam.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Poznáš/používal si Atomic Variables? Akú majú výhodu?**  
Atomickejšie premenné v Jave poskytujú spôsob, ako vykonávať atomické operácie na jednotlivých premenných bez potreby použitia **synchronized** bloku alebo metód. Tieto premenné sú súčasťou balíka **java.util.concurrent.atomic**, ktorý obsahuje triedy ako **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, a mnohé ďalšie. Tieto atomickejšie premenné sú navrhnuté tak, aby zjednodušili vývoj konkurentných aplikácií tým, že zabezpečia "lock-free" prístup k spoločným premenným.

**Výhody atomickejších premenných**

1. **Bezpečnosť vlákien (Thread Safety):** Atomickejšie premenné zaručujú, že operácie na nich sú atomické, čo znamená, že každá operácia je úplne dokončená pred začiatkom ďalšej operácie, bez možnosti prerušenia. Toto eliminuje súbehové chyby v kontextoch s viacerými vláknami.
2. **Vysoký výkon:** Keďže atomickejšie operácie sa vykonávajú bez použitia zámkov, aplikácie môžu dosiahnuť vyššiu úroveň súbežnosti. Použitie zámkov môže spôsobiť blokovanie a prepnutie kontextu vlákien, čo znižuje výkon. Atomickejšie premenné tieto problémy obchádzajú poskytovaním rýchlejších a efektívnejších operácií.
3. **Jednoduchosť použitia:** Kód s atomickejšími premennými je často jednoduchší a čistejší ako kód používajúci explicitné zámky alebo synchronizované bloky. Toto môže zlepšiť čitateľnosť kódu a uľahčiť správu a údržbu.
4. **Lock-free programovanie:** Atomickejšie premenné umožňujú "lock-free" programovanie, čo môže zlepšiť škálovateľnosť a výkon aplikácií tým, že minimalizuje blokovanie a čakanie vlákien.

**Príklady operácií s atomickejšími premennými**

* **Aktualizácie hodnôt:** Metódy ako **getAndIncrement()**, **getAndDecrement()**, **getAndAdd()**, **incrementAndGet()**, **decrementAndGet()**, a **addAndGet()** umožňujú atomické aktualizácie hodnôt bez potreby synchronizácie.
* **Porovnávanie a nastavovanie:** Metóda **compareAndSet(expectedValue, updateValue)** atomicky aktualizuje hodnotu na **updateValue**, iba ak aktuálna hodnota je rovná **expectedValue**.

Atomickejšie premenné sú teda mocným nástrojom pre vývojárov, ktorí potrebujú zabezpečiť bezpečnosť vlákien a vysoký výkon v konkurentných aplikáciách, a to všetko s menšou námahou a komplexnosťou ako pri používaní tradičných zámkov a synchronizovaných blokov.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké SCOPEs majú Spring BEANs?**  
Spring framework poskytuje rôzne oblasti pôsobnosti (scopes) pre beany, ktoré určujú, ako a kedy sa inštancia bean-u vytvára, zdieľa a zničí. Každý scope má svoje vlastné použitie a vhodnosť pre rôzne scenáre. Tu sú najčastejšie používané scopes pre Spring beany:

1. **Singleton** (predvolený): V rámci jednej Spring IoC kontajnera sa pre každý bean definovaný ako singleton vytvorí práve jedna inštancia. Toto je predvolený scope. Singleton bean je vytvorený pri štarte kontajnera a zničený pri jeho zatvorení, a je zdieľaný medzi všetkými, ktorí požadujú referenciu na tento bean.
2. **Prototype**: Každý požiadavok na bean s týmto scopom vytvorí novú inštanciu bean-u. To znamená, že ak dva rôzne beany alebo ten istý bean viackrát požadujú inštanciu prototype bean-u, každý požiadavok dostane unikátnu inštanciu.
3. **Request**: Tento scope je špecifický pre webové aplikácie. Pre každú HTTP požiadavku sa vytvorí nová inštancia bean-u. Bean definovaný s týmto scopom bude mať jednu inštanciu pre každú HTTP požiadavku.
4. **Session**: Rovnako špecifický pre webové aplikácie, session scope znamená, že Spring vytvorí inštanciu bean-u pre každú HTTP session. To umožňuje udržiavať stav bean-u počas celej session.
5. **Application**: Bean s týmto scopom je špecifický pre celú webovú aplikáciu. Spring vytvorí jednu inštanciu bean-u pre celý ServletContext. To je užitočné pre dáta, ktoré sú globálne a majú byť zdieľané medzi všetkými sessionmi a požiadavkami v aplikácii.
6. **Websocket**: Tento scope je špecifický pre webové websocket spojenia. Podobne ako request a session scopes, ale špecificky pre životný cyklus websocket spojenia.

Každý z týchto scopes má svoje vlastné použitie a umožňuje vývojárom prispôsobiť správanie Spring bean-ov podľa potrieb aplikácie. Voľba správneho scopu je kľúčová pre efektívne spravovanie pamäte a správanie aplikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Máš skúsenosti s optimalizáciou a performance tuningom (HEAPy, GARBAGE COLLECTOR)?**

**Správa HEAP pamäte**

* **Monitorovanie a Analýza HEAPu:** Použitie nástrojov ako VisualVM, JProfiler, alebo GC log analyzátory môže pomôcť identifikovať použitie pamäte a miesta, kde dochádza k únikom pamäti.
* **Optimalizácia Veľkosti HEAPu:** Správne nastavenie veľkosti počiatočnej (**-Xms**) a maximálnej (**-Xmx**) haldy môže zlepšiť výkon tým, že sa zníži počet GC cyklov a zároveň sa zabráni vypĺňaniu pamäte.

**Tuning Garbage Collectora**

* **Výber Správneho GC:** Java poskytuje viacero typov Garbage Collectorov (napr. Serial, Parallel, CMS, G1, ZGC, Shenandoah), a výber závisí od potrieb aplikácie a charakteristík nasadenia.
* **Konfigurácia GC:** Afinovanie parametrov GC, ako sú veľkosti generácií (Young, Old), môže značne zlepšiť výkon aplikácie. Znižovanie doby pauzy GC alebo optimalizácia preputovania (throughput) sú bežné ciele.

**Optimalizácia Kódu**

* **Identifikácia a Oprava Únikov Pamäte:** Vyšetrenie a oprava únikov pamäte, ktoré môžu spôsobiť, že aplikácia spotrebuje viac pamäti, než je nevyhnutné.
* **Efektívna Práca s Kolekciami:** Použitie vhodnej dátovej štruktúry pre danú situáciu a zabezpečenie, aby kolekcie neboli zbytočne veľké.

**Profilovanie a Benchmarking**

* **Použitie Profilovacích Nástrojov:** Profilovanie aplikácie počas vývoja aj v produkčnom prostredí pomáha identifikovať úzke miesta výkonu.
* **Benchmarking:** Použitie nástrojov ako JMH (Java Microbenchmark Harness) pre presné meranie výkonnosti kritických častí kódu.

**Best Practices**

* **Optimalizácia na Správnej Úrovni:** Dôležité je vykonávať optimalizácie s porozumením reálnemu vplyvu na aplikáciu, aby sa predišlo "premature optimization".
* **Požitie Asynchronného a Reaktívneho Programovania:** Môže zlepšiť využitie systémových zdrojov a celkovú odozvu aplikácie.

Každá z týchto oblastí vyžaduje dôkladné pochopenie fungovania JVM, ako aj schopnosť analyzovať a interpretovať relevantné metriky a logy. Vždy je dôležité vyvážiť potrebu optimalizácie s časom a zdrojmi, ktoré sú k dispozícii na jej dosiahnutie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aké zaujímavé features z posledných verzií JAVA poznáš?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Poznáš funkciu SWITCH?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Vieš opísať JAVA pamäte?**

Java používa spravovanú pamäť, čo znamená, že vývojári nemusia priamo spravovať alokáciu a dealokáciu pamäte. Java Virtual Machine (JVM) spravuje pamäť automaticky, a to najmä prostredníctvom garbage collection, ktorá uvoľňuje pamäť, keď objekty už nie sú potrebné. Pamäť v JVM je rozdelená do niekoľkých oblastí:

1. **Heap Memory:**
   * Toto je hlavná oblasť pamäte, kde Java ukladá objekty a inštancie tried.
   * Heap sa ďalej delí na Young Generation (Eden space, Survivor spaces) a Old Generation (Tenured space).
     + **Young Generation:** Všetky nové objekty sú vytvorené tu a väčšina objektov tu rýchlo zomiera.
     + **Eden Space:** Tu sú umiestnené všetky novovytvorené objekty.
     + **Survivor Spaces (S0 a S1):** Objekty, ktoré prežili garbage collection v Eden space, sú presunuté do jedného z týchto priestorov.
     + **Old Generation:** Objekty, ktoré prežili viacero garbage collection cyklov v young generation, sú presunuté do tejto oblasti.
   * Keď je heap plný, garbage collector sa spustí a uvoľní pamäť odstránením objektov, ktoré už nie sú dosiahnuteľné.
2. **Non-Heap Memory:**
   * Táto časť pamäte je používaná JVM na ukladanie metadát tried, konštánt a statických členov.
   * Delí sa na Metaspace (v starších verziách Permanent Generation alebo PermGen), Code Cache a Compressed Class Space.
     + **Metaspace:** Ukladá štruktúry ako triedy a metódy.
     + **Code Cache:** Uchováva kompilovaný a optimalizovaný kód.
     + **Compressed Class Space:** Uchováva triedy v komprimovanej forme, ak je to povolené.
3. **Stack Memory:**
   * Pre každé vlákno JVM vytvorí zásobník (stack), ktorý ukladá lokálne premenné a informácie o volaniach metód.
   * Každé volanie metódy vytvorí rámec zásobníka (stack frame) s lokálnymi premennými a referenciami na iné objekty v heap.
4. **Direct Memory (Priama pamäť):**
   * Táto časť pamäte sa využíva mimo Java heap a je prístupná prostredníctvom ByteBuffer. Je to užitočné pre veľké alebo dlhotrvajúce buffery, ktoré by mohli zbytočne zaťažiť garbage collector.

Správa pamäte je kľúčová pre výkon Java aplikácií, a preto je dôležité rozumieť, ako tieto rôzne oblasti pamäte fungujú a ako môžu ovplyvniť vašu aplikáciu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Z čoho sa skladá JAVA MEMORY HEAP?**

Samozrejme, Java Memory Heap je oblasť pamäti, kde Java runtime ukladá vytvorené objekty a inštancie tried. Heap je dynamicky alokovaný pri štarte JVM a môže sa meniť počas behu aplikácie. Skladá sa z niekoľkých generácií, ktoré umožňujú efektívnejšie spravovať pamäť a garbage collection. Tu je jednoduchý popis komponentov Java Memory Heap:

1. **Young Generation:**
   * Táto časť heapu je miesto, kde sa vytvárajú všetky nové objekty. Je relatívne malá, pretože sa predpokladá, že väčšina objektov bude "krátkoživých".
   * **Eden Space:** Keď vytvoríte nový objekt, je pravdepodobne umiestnený práve tu. Je to najväčšia časť young generation.
   * **Survivor Spaces (S0 a S1):** Po každom spustení garbage collection v young generation, objekty, ktoré prežili, sú presunuté do jedného z týchto priestorov. Pri ďalšom behu garbage collection sa preživšie objekty presunú z jedného survivor space do druhého.
2. **Old Generation (Tenured Generation):**
   * Objekty, ktoré prežili viacero cyklov garbage collection v young generation, sú presunuté do tejto časti heapu. Táto časť je väčšia a ukladá objekty, ktoré existujú dlhšiu dobu.
   * Garbage collection v old generation (známa ako Major GC alebo Full GC) je zvyčajne menej častá, ale trvá dlhšie, pretože prehľadáva a čistí väčšie množstvo pamäte.

Garbage collector v Jave pracuje na princípe dosiahnuteľnosti objektov. Ak sa k objektu nedá dostať žiadnou živou cestou z "koreňov" aplikácie (napr. lokálne premenné metód a statické premenné tried), považuje sa za neprístupný a jeho pamäť môže byť uvoľnená.

Je dôležité si uvedomiť, že nadmerné vytváranie objektov a nedostatočné uvoľňovanie pamäte môže viesť k častým spusteniam garbage collection, čo môže negatívne ovplyvniť výkon aplikácie. Správne návrhové vzory a pamäťová optimalizácia sú kľúčové pre efektívnu prácu s Java heap pamäťou.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako si riešil MEMORY LEAK?**

Riešenie memory leak (únik pamäte) v Java aplikácii si vyžaduje systematický prístup k identifikácii, diagnostikovaniu a následnému odstraňovaniu problému. Tu je postup, ktorý by som odporúčal nasledovať:

1. **Identifikácia problému:**
   * **Monitorovanie:** Použite nástroje na monitorovanie využitia pamäte vašej aplikácie. Ak vidíte, že pamäť neustále rastie a nie je uvoľnená po garbage collection, môže to byť indikácia memory leak.
   * **Profiling:** Profilery ako VisualVM, YourKit alebo JProfiler vám môžu pomôcť identifikovať, kde sa pamäť používa. Profilery umožňujú sledovať alokáciu objektov a môžu vám ukázať, ktoré objekty zostávajú v pamäti dlhšie, než by mali.
2. **Analýza a diagnostika:**
   * **Heap dump analýza:** Vygenerujte heap dump a analyzujte ho pomocou nástrojov ako Eclipse Memory Analyzer (MAT) alebo VisualVM. Hľadajte objekty, ktoré majú neočakávane veľký počet inštancií alebo ktoré zaberajú veľa pamäte.
   * **Kontrola kódu:** Skontrolujte časti kódu, ktoré súvisia s týmito objektmi. Hľadajte bežné príčiny memory leakov, ako sú statické kolekcie, listenery, nesprávne použitie cache alebo neukončené externé zdroje a vlákna.
3. **Riešenie problému:**
   * **Oprava kódu:** Keď identifikujete zdroj úniku, opravte problém v kóde. To môže zahŕňať odstránenie nepotrebných objektov z kolekcií, správne zatváranie zdrojov, odregistrovanie listenerov atď.
   * **Optimalizácia:** Zvážte použitie slabých odkazov (WeakReferences) alebo SoftReferences, ak je to vhodné, na zníženie rizika memory leak.
   * **Revízia a testovanie:** Po oprave úniku pamäte znovu otestujte aplikáciu, aby ste sa uistili, že problém bol vyriešený a nezaviedli ste nové problémy.
4. **Prevencia pre budúcnosť:**
   * **Kódové preskúmania:** Pravidelne vykonávajte kódové preskúmania na identifikáciu potenciálnych problémov s pamäťou skôr, než sa dostanú do produkcie.
   * **Využívanie nástrojov:** Integrujte nástroje na profilovanie a monitorovanie pamäte do vývojového a testovacieho procesu, aby ste mohli rýchlo identifikovať a riešiť akékoľvek problémy s pamäťou.
   * **Odborné vzdelávanie:** Vzdelávajte svoj tím o najlepších postupoch pre správu pamäte v Jave, aby mohli písať efektívnejší kód a lepšie rozumieť potenciálnym problémom s pamäťou.

Riešenie memory leak je často iteratívny proces, kde analýza a opravy môžu vyžadovať niekoľko cyklov na úplné odstránenie problému.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Rozdiel medzi NOD, POD a SERVICE v Kubernetes?**

V Kubernetes, Node (Nod), Pod a Service sú základné komponenty, ktoré spolu pracujú na nasadení a správe kontajnerizovaných aplikácií. Tu je základný rozdiel medzi nimi:

1. **Node (Nod):**
   * Node je fyzický alebo virtuálny stroj, na ktorom beží Kubernetes. Je to pracovná jednotka, ktorá obsahuje služby nevyhnutné na spustenie Podov.
   * Každý Node obsahuje Kubelet, ktorý je agentom spravujúcim Pod-y a komunikujúcim s Kubernetes masterom.
   * Node-y poskytujú zdroje (CPU, pamäť, sieťové rozhrania, atď.) pre kontajnery v Pod-och.
2. **Pod:**
   * Pod je základná nasadzovacia jednotka v Kubernetes, ktorá obsahuje jeden alebo viac kontajnerov, ktoré sú na sieťovej úrovni spoločné a môžu zdieľať úložisko.
   * Všetky kontajnery v jednom Pode sú naplánované na ten istý Node a zdieľajú rovnaký kontext (napr. IP adresu, porty, atď.).
   * Pod-y sú efemérne a nemajú trvalý životný cyklus – v prípade zlyhania sú nahradené novými identickými Pod-mi.
3. **Service:**
   * Service je abstrakcia, ktorá definuje logický set Pod-ov a politiku prístupu k nim. Tento prístup je často poskytovaný prostredníctvom sieťovej služby, ako je load balancer.
   * Service umožňuje komunikáciu medzi rôznymi komponentami vnútri clustera alebo z externých zdrojov s Pod-mi bez toho, aby bolo potrebné vedieť, na akých Node-och sú Pod-y umiestnené.
   * Keď Pod-y prichádzajú a odchádzajú, Service zabezpečuje, že sieťové spojenie zostáva konzistentné, presmerovaním sieťovej komunikácie podľa potreby.

Zjednodušene povedané, Node je stroj, ktorý spúšťa kontajnery, Pod je najmenšia nasadzovacia jednotka, ktorá môže obsahovať jeden alebo viacero kontajnerov, a Service definuje, ako sú tieto Pod-y dostupné v rámci siete.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Co je to Kubernetis ?**  
Kubernetes a Docker sú súvisiace, ale slúžia odlišným účelom a často sa používajú spoločne v kontajnerizovaných prostrediach. Poďme si objasniť ich rozdiely a vzťahy:

1. **Docker:**
   * Docker je nástroj na vytváranie, distribúciu a spúšťanie kontajnerov. Umožňuje balenie aplikácie a jej závislostí do štandardizovaného jednotkového kontajnera, ktorý môže byť spustený na akomkoľvek Linuxovom alebo Windows hostiteli, ktorý podporuje Docker.
   * Docker poskytuje jednoduchý a efektívny spôsob, ako izolovať prostredie aplikácie od podkladového systému, zabezpečujúc konzistentnosť naprieč rôznymi vývojovými a produkčnými prostrediami.
   * Docker v sebe zahŕňa Docker Engine, ktorý je runtime prostredím pre kontajnery, a Docker Hub, ktorý je cloudovou službou pre zdieľanie a distribúciu kontajnerových obrazov.
2. **Kubernetes:**
   * Na druhej strane, Kubernetes je orchesteráčný systém na správu kontajnerov na veľkú škálu. Umožňuje automatizovať nasadenie, škálovanie a prevádzku kontajnerizovaných aplikácií.
   * Kubernetes nevytvára kontajnery; namiesto toho ich organizuje a spravuje. Môže pracovať s kontajnermi vytvorenými pomocou Dockeru alebo iných kontajnerových technológií, ako je napríklad rkt alebo CRI-O.
   * Kubernetes poskytuje platformu pre automatizovanú inštaláciu, škálovanie a správu kontajnerových aplikácií, a to v rozsiahlych, distribuovaných prostrediach.

Keď hovoríme, že "používame Kubernetes", zvyčajne to znamená, že používame Kubernetes na koordináciu a správu aplikácií, ktoré bežia vo vnútri kontajnerov, často vytvorených pomocou Dockeru. Teda, Kubernetes a Docker sa veľmi často používajú spoločne, ale slúžia rôznym účelom: Docker ako nástroj na vytváranie a spúšťanie kontajnerov, Kubernetes ako systém na ich orchesteráciu.---------

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

**CONSUMER OFFSET je na čo?**

Consumer offset v kontexte správy správ, ako je napríklad Apache Kafka, je kľúčovým pojmom pre pochopenie, ako spotrebitelia (consumers) spracúvajú správy v témach (topics). Tu je podrobnejšie vysvetlenie a kontext použitia:

1. **Čo je Consumer Offset?**
   * Consumer offset je číselná hodnota, ktorá udáva pozíciu v rámci témy (topic), odkiaľ má spotrebiteľ (consumer) začať čítať správy. V systéme ako Kafka, každá správa v téme má svoje jedinečné sekvenčné ID známe ako offset.
   * Offset umožňuje spotrebiteľom vedieť, ktoré správy už boli spracované a odkiaľ pokračovať v prípade reštartu alebo výpadku.
2. **Kedy a na čo sa používa?**
   * **Spracovanie správ:** Keď spotrebiteľ číta správy z témy, udržuje si informáciu o poslednom spracovanom offsete, takže vie, ktorú správu má čítať ďalej.
   * **Zabezpečenie odolnosti:** V prípade, že spotrebiteľ alebo spotrebiteľská skupina zlyhá alebo je reštartovaná, môžu pokračovať v čítaní správ z posledného uloženého offsetu, čím sa zabráni strate alebo duplikácii správ.
   * **Paralelizácia a škálovanie:** Rôzne spotrebiteľské inštancie v skupine môžu čítať rôzne partície rovnakej témy paralelne, pričom každá si udržuje vlastný offset, čo umožňuje efektívne škálovanie spracovania správ.
3. **Ako sa udržuje?**
   * V Apache Kafka, offsety sa ukladajú v špeciálnej téme známej ako **\_\_consumer\_offsets**, kde každá spotrebiteľská skupina udržiava svoje offsety, čo umožňuje spotrebiteľom reštartovať alebo obnoviť svoj stav.

Zhrnuté, consumer offset je kritický pre správne spracovanie správ v distribuovaných správových systémoch, umožňuje odolnosť proti chybám, zabezpečuje konzistentnosť spracovania a podporuje horizontálne škálovanie spotrebiťelských aplikácií.

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Rozdiel medzi Kafka a RabbitMQ:**  
Apache Kafka a RabbitMQ sú oba populárne systémy správ (message brokers), ale sú navrhnuté na trochu odlišné použitie a majú rôzne charakteristiky:

1. **Základné rozdiely:**
   * **Apache Kafka:** Je distribuovaný streamovací platforma, ktorá je optimalizovaná pre vysoký prietok a ukladanie veľkého množstva dát. Kafka je vhodná pre situácie, kde potrebujete spracovávať veľké objemy dát v reálnom čase. Umožňuje publikovanie a odber správ medzi vydavateľmi a spotrebiteľmi na báze tém (topics). Kafka je navrhnutá tak, aby efektívne spracovala prúdy správ a uchovávala ich po dlhú dobu.
   * **RabbitMQ:** Je to tradičný message broker, ktorý implementuje pokročilé správanie spracovania správ, vrátane smerovania, záťaže a potvrdenia správ. RabbitMQ je vhodný pre komplexné smerovacie schémy a poskytuje rôzne typy výmenníkov (exchanges), ktoré umožňujú presné smerovanie a distribúciu správ. Je vhodný pre situácie, kde potrebujete zabezpečiť spoľahlivú komunikáciu medzi komponentmi v distribuovaných systémoch.
2. **Use Cases:**
   * **Kafka:** Je často používaná pre logy udalostí, monitorovanie transakcií v reálnom čase, streamovanie dát, a v situáciách, kde je potrebné spracovávať a analyzovať veľké objemy dát vysokou rýchlosťou.
   * **RabbitMQ:** Je vhodný pre úlohy, kde je potrebná zaručená doručiteľnosť správ, komplexné smerovanie, spracovanie rôznych úrovní prioritizácie správ a kde sú požiadavky na objem dát nižšie, ale je potrebná vyššia flexibilita v smerovaní správ.
3. **Vlastnosti a architektúra:**
   * **Kafka:** Ukladá správy vo forme logov v témach, ktoré sú rozdelené do partícií a replikované medzi servery pre zvýšenie odolnosti a prietoku. Spotrebitelia čítajú správy v poradí, v akom boli uložené, a môžu kontrolovať svoj vlastný offset.
   * **RabbitMQ:** Poskytuje širokú škálu možností smerovania vrátane direct, topic, headers a fanout výmenníkov, ktoré umožňujú veľmi flexibilné a sofistikované spracovanie správ. RabbitMQ ukladá správy do frontov a zabezpečuje, že sú správne doručené a potvrdené.

Zhrnutie: Výber medzi Kafka a RabbitMQ závisí od konkrétnych požiadaviek vašej aplikácie. Ak potrebujete spracovať veľké množstvá dát alebo prúdy správ v reálnom čase, Kafka je pravdepodobne lepšia voľba. Ak potrebujete zložité smerovanie, flexibilnú konfiguráciu a zabezpečenú komunikáciu, RabbitMQ by mohlo byť vhodnejšie riešenie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**ACTUATOR v SPRING je na čo?**

Actuator v Spring Boot je nástroj, ktorý poskytuje súbor služieb a endpointov umožňujúcich monitorovanie a správu vašej aplikácie. Spring Boot Actuator je integrovaný do Spring Boot aplikácií a poskytuje cenné informácie o stave a fungovaní aplikácie v produkčnom prostredí.

Tu sú niektoré kľúčové funkcie a využitia Spring Boot Actuator:

1. **Monitorovanie aplikácie:** Actuator poskytuje rôzne endpointy, ktoré môžu byť použité na získanie informácií o behu aplikácie, ako sú metriky, zdravie, informácie o prostredí, konfigurácie bean a ďalšie.
2. **Zdravotné kontroly (Health checks):** Actuator poskytuje endpoint **/health**, ktorý ukazuje rôzne zdravotné indikátory aplikácie, ako sú stav databáze, stav cache, stav správy frontov a ďalšie.
3. **Prístup k metrikám:** Endpoint **/metrics** poskytuje rôzne užitočné metriky aplikácie, ako sú počty a trvanie HTTP požiadaviek, stav systémových metrík (napr. využitie pamäte, využitie procesora) a ďalšie.
4. **Auditovanie:** Actuator môže zaznamenávať auditné udalosti, ako sú prihlásenie a odhlásenie užívateľa, a poskytovať ich prostredníctvom endpointu **/auditevents**.
5. **Podrobné informácie o konfigurácii:** Actuator poskytuje detailné informácie o konfigurácii Spring aplikácie prostredníctvom endpointov ako **/configprops** alebo **/env**.
6. **Správa a kontrola logovania:** Actuator umožňuje zobraziť a zmeniť konfiguráciu logovania v aplikácii naživo prostredníctvom endpointov ako **/loggers**.
7. **Zastavenie aplikácie:** Actuator poskytuje endpoint **/shutdown**, ktorý môže byť použitý na bezpečné zastavenie aplikácie, ak je to povolené.

Actuator je dôležitý nástroj pre vývojárov a prevádzkových inžinierov, pretože poskytuje cenné vnútorné informácie, ktoré môžu byť použité na monitorovanie, diagnostiku a správu aplikácií postavených na Spring Boot.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Máš skúsenosť s DOKUMENTÁCIOU DIZAJNOV?**

Dokumentácia dizajnov je kritickým krokom vo vývoji softvéru, produktov, architektúry alebo akéhokoľvek inžinierskeho projektu. Umožňuje tímom zachovať konzistentnosť v projekte, efektívne komunikovať medzi rôznymi členmi tímu a zabezpečiť, že všetky aspekty dizajnu sú dobre pochopené a dokumentované pre aktuálnu implementáciu aj budúce referencie. Tu je niekoľko kľúčových aspektov, ktoré by mala obsahovať efektívna dokumentácia dizajnov:

1. **Úvod a Kontext:**
   * Popis projektu a dizajnu, vrátane cieľov a očakávaných výsledkov.
   * Kontext alebo problém, ktorý dizajn rieši.
2. **Dizajnové Špecifikácie:**
   * Podrobné informácie o dizajne vrátane všetkých relevantných vlastností, parametrov, a prvkov dizajnu.
   * Diagramy, obrázky, alebo iné vizuálne reprezentácie, ktoré pomáhajú ilustrovať dizajn.
3. **Princípy a Východiská:**
   * Zásady a predpoklady, na ktorých je dizajn založený.
   * Odkazy na štandardy alebo best practices, ktoré dizajn sleduje.
4. **Podrobnosti Implementácie:**
   * Návody alebo pokyny pre implementáciu dizajnu.
   * Informácie o tom, ako dizajn interaguje s ostatnými časťami systému alebo produktu.
5. **Testovanie a Validácia:**
   * Metódy a prístupy k testovaniu dizajnu.
   * Scenáre alebo prípady použitia, ktoré by mali byť otestované.
6. **Verzovanie a Revízie:**
   * Informácie o verzii dizajnu a histórii revízií.
   * Zmeny, ktoré boli v každej verzii urobené.
7. **Referencie:**
   * Odkazy na externé dokumenty, zdroje alebo iné dizajny, ktoré sú relevantné alebo boli použité ako inšpirácia.

Dokumentácia dizajnov by mala byť živý dokument, ktorý sa aktualizuje a udržiava počas celého životného cyklu dizajnu a vývoja. Toto je kľúčové pre zachovanie relevancie a užitočnosti dokumentácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Čo je DDD?**  
DDD, alebo Domain-Driven Design, je prístup k softvérovému dizajnu a vývoju zameraný na komplexné potreby domény, pre ktorú sa softvér vytvára. Tento prístup zdôrazňuje pochopenie domény, s ktorou pracujete, a vývoj softvéru, ktorý túto doménu čo najpresnejšie odráža.

Kľúčové aspekty DDD zahŕňajú:

1. **Ubiquitous Language (Všadeprítomný jazyk):** DDD zdôrazňuje význam komunikácie a vytvárania spoločného jazyka medzi vývojármi a odborníkmi na doménu. Tento jazyk by mal byť konzistentne používaný vo všetkých aspektoch projektu, od analýzy poznatkov až po implementáciu.
2. **Domain Model (Doménový model):** Srdcom DDD je bohatý a expresívny doménový model, ktorý zachytáva kľúčové aspekty domény a pravidlá. Model by mal byť orientovaný objektovo a odrážať skutočné entity a vzťahy, ktoré existujú v doméne.
3. **Bounded Contexts (Ohraničené kontexty):** Doména sa často delí na menšie, spravovateľné časti známe ako ohraničené kontexty. Každý kontext má svoj vlastný doménový model a všadeprítomný jazyk, a je navrhnutý tak, aby minimalizoval závislosti na iných kontextoch.
4. **Context Mapping:** Tento proces identifikuje a mapuje vzťahy medzi rôznymi ohraničenými kontextmi v systéme, čo pomáha pochopiť, ako by mali spolu interagovať.
5. **Entities and Value Objects:** DDD rozlišuje medzi entitami, ktoré majú jedinečnú identitu a kontinuálne existujú cez čas, a hodnotovými objektmi, ktoré sú definované len svojimi atribútmi a nemajú identitu.
6. **Aggregates:** Aggregate je skupina objektov, ktoré sa spoločne spravujú ako jednotka pre účely údajovej zmeny. Každý agregát má koreňový objekt, známy ako aggregate root, cez ktorý sa vykonáva všetka interakcia s agregátom.
7. **Repositories and Services:** DDD definuje repozitáre, ktoré sú zodpovedné za ukladanie a obnovovanie agregátov, a služby, ktoré implementujú doménovú logiku, ktorá nie je prirodzene súčasťou entity alebo hodnotového objektu.

DDD je významným prístupom v softvérovom inžinierstve, ktorý pomáha riešiť komplexné problémy vytváraním bohatých, dobre štruktúrovaných modelov založených na hlbokom pochopení domény.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi virtualizáciou a kontajnerizáciou?**

Predstavte si, že máte veľkú kanceláriu plnú pracovných staníc (počítačov). Každá pracovná stanica je ako samostatný virtuálny stroj v prostredí virtualizácie – má svoj vlastný stôl, kreslo, počítač, klávesnicu, myš a tak ďalej. Aj keď sú všetky v jednej kancelárii (na jednom fyzickom serveri), každá z nich pracuje nezávisle a má všetko, čo potrebuje, aby mohla fungovať ako samostatné pracovisko.

**Virtualizácia:** Je ako mať individuálne pracovné stanice v kancelárii. Ak by ste chceli pridať ďalšiu pracovnú stanicu, potrebovali by ste pridať ďalší stôl, kreslo a počítačový hardvér. Každá pracovná stanica je izolovaná; ak jedna zlyhá alebo potrebuje reštart, ostatné nie sú ovplyvnené.

**Príklad s virtualizáciou:**

* Máte tri virtuálne stroje: jeden beží na Windows, druhý na Linuxe a tretí tiež na Linuxe, ale každý s rôznymi aplikáciami a službami.
* Každý virtuálny stroj používa vlastné zdroje (RAM, CPU) nezávisle na ostatných.

Teraz si predstavte, že namiesto individuálnych pracovných staníc máte veľkú pracovnú plochu, kde každý má svoj priestor s vlastnou klávesnicou a monitorom, ale zdieľa jeden veľký stôl a počítač.

**Kontajnerizácia:** Je ako zdieľaný veľký stôl, kde každý má svoj vlastný priestor. Ak pridáte ďalšiu osobu, jednoducho jej pridelíte ďalší priestor na tom istom stole. Všetci používajú rovnaký počítač (operačný systém), ale majú svoje vlastné oblasti, kde môžu pracovať nezávisle na ostatných.

**Príklad s kontajnerizáciou:**

* Máte server (veľký stôl), kde beží viac kontajnerov (rôzne oblasti na stole).
* Jeden kontajner môže bežať webový server, ďalší databázový server a tretí môže vykonávať nejaký výpočet.
* Všetky kontajnery zdieľajú rovnaký operačný systém (podobne ako zdieľanie veľkého počítača/stola), ale každý kontajner je izolovaný a má vlastné softvérové knižnice a aplikácie.

Zhrnutie: Virtualizácia je ako mať samostatné pracovné stanice pre každú osobu, zatiaľ čo kontajnerizácia je ako zdieľaný veľký stôl, kde každý má svoj vlastný priestor, ale niektoré zdroje sú spoločné. Kontajnerizácia je efektívnejšia, keďže lepšie využíva dostupné zdroje a umožňuje rýchlejšie spustenie a zastavenie aplikácií.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi autentifikáciou a autorizáciou?**

Autentifikácia a autorizácia sú dva základné bezpečnostné procesy, ktoré sa často používajú spoločne, ale majú odlišné úlohy:

1. **Autentifikácia:**
   * **Čo to je:** Autentifikácia je proces overovania identity používateľa. Cieľom je zistiť, či je používateľ skutočne tým, kým tvrdí, že je.
   * **Ako to funguje:** Pri autentifikácii používateľ poskytne svoje poverenia (napr. meno a heslo, biometrické údaje, jednorazové heslo atď.), ktoré systém porovná s uloženými údajmi. Ak sa poverenia zhodujú, používateľ je autentifikovaný.
   * **Príklad:** Prihlásenie do e-mailového účtu pomocou mena a hesla.
2. **Autorizácia:**
   * **Čo to je:** Autorizácia je proces udelenia alebo zamietnutia práv používateľovi na vykonanie určitých akcií v systéme po tom, čo bol úspešne autentifikovaný.
   * **Ako to funguje:** Po autentifikácii systém určí, aké akcie môže používateľ vykonávať, na základe pridelených oprávnení alebo rolí. Autorizácia určuje, ku ktorým zdrojom a operáciám má používateľ prístup.
   * **Príklad:** Po prihlásení do e-mailového účtu (autentifikácia) máte možnosť čítať správy, posielať nové e-maily alebo mazať existujúce e-maily (autorizácia).

**Zhrnutie:** Autentifikácia je teda "overenie totožnosti" (kto ste?), zatiaľ čo autorizácia je "udelenie prístupových práv" (čo môžete robiť?). V praxi sa autentifikácia vždy deje pred autorizáciou a je nevyhnutná na zabezpečenie toho, aby boli oprávnenia pridelené správnemu používateľovi.

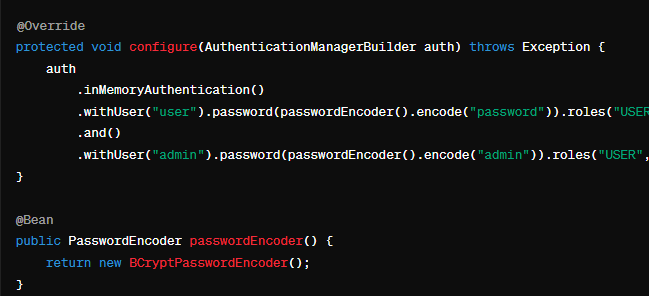
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako funguje autentifikácia a autorizácia v Springu?**  
V rámci Spring frameworku, špeciálne v Spring Security, autentifikácia a autorizácia sú kľúčové komponenty, ktoré zabezpečujú, že len oprávnení používatelia majú prístup k určitým zdrojom alebo môžu vykonávať určité akcie. Tu je, ako tieto procesy fungujú v Springu:

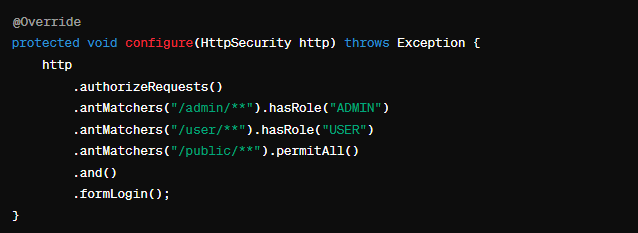
1. **Autentifikácia:**
   * V Spring Security, autentifikačný proces začína, keď používateľ podá autentifikačné poverenia, obvykle prostredníctvom prihlasovacieho formulára.
   * Tieto poverenia sú odoslané na server, kde **AuthenticationManager** spracuje autentifikačný požiadavok.
   * **AuthenticationManager** používa **AuthenticationProvider**, ktorý overí poverenia používateľa proti úložisku používateľov (napr. databáze, LDAP).
   * Po úspešnej autentifikácii sa vytvorí **Authentication** objekt, ktorý obsahuje identitu a oprávnenia používateľa a je uložený v **SecurityContext**.
2. **Autorizácia:**
   * Po autentifikácii prichádza proces autorizácie, ktorý rozhoduje, či má autentifikovaný používateľ prístup k určitým zdrojom alebo môže vykonať určité akcie.
   * V Spring Security sa to zvyčajne kontroluje pomocou anotácií (napr. **@PreAuthorize**, **@PostAuthorize**, **@Secured**) alebo konfiguráciou zabezpečenia HTTP, kde môžete špecifikovať prístupové pravidlá založené na URL, rolách alebo iných atribútoch.
   * Spring Security overí, či **Authentication** objekt v **SecurityContext** obsahuje potrebné oprávnenia (udelené úlohy/role alebo iné bezpečnostné atribúty) na prístup k požadovaným zdrojom alebo vykonanie akcie.

**Príklad autentifikácie a autorizácie v Spring Security:**

Autentifikácia (v konfiguračnej triede):



Autorizácia (v konfiguračnej triede):



V tomto príklade konfigurácia zabezpečenia vymedzuje, že cesty začínajúce **/admin/** vyžadujú, aby mal používateľ rolu **ADMIN**, cesty začínajúce **/user/** vyžadujú rolu **USER**, a cesty začínajúce **/public/** sú prístupné všetkým bez autentifikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké patterny špecifické pre microservices poznáš/si používal?**  
V architektúre založenej na mikroslužbách sa používa rad špecifických vzorov, ktoré riešia bežné problémy spojené s dizajnom, komunikáciou, správou dát a odolnosťou v distribuovanom prostredí. Tu je niekoľko kľúčových vzorov špecifických pre mikroslužby, ktoré sú široko používané:

1. **API Gateway:**
   * API Gateway je jednotný vstupný bod pre všetky klientové požiadavky, ktoré smeruje do rôznych mikroslužieb. Zjednodušuje komplexitu klientov tým, že agreguje rôzne individuálne endpointy mikroslužieb do jedného rozhrania.
2. **Service Discovery:**
   * V prostredí mikroslužieb, kde služby dynamicky prichádzajú a odchádzajú, potrebujú služby spôsob, ako efektívne nájsť a komunikovať s inými službami. Service Discovery umožňuje službám automaticky zistiť a komunikovať s inými službami v sieti.
3. **Circuit Breaker:**
   * Tento vzor zabraňuje rozšíreniu zlyhania v jednej službe na iné služby. Ak služba zlyháva alebo je príliš pomalá, obvodový spínač preruší volanie k tejto službe na určitý čas, čím znižuje záťaž na zlyhávajúcu službu a udržuje stabilitu systému.
4. **Config Server:**
   * Centralizovaný konfiguračný server umožňuje spravovať konfiguračné nastavenia pre všetky mikroslužby na jednom mieste. To uľahčuje správu a zabezpečuje konzistentnosť konfigurácií naprieč službami.
5. **Aggregator:**
   * Aggregator je vzor, ktorý kompozituje výsledky z viacerých služieb do jednej agregovanej odpovede. Toto je obzvlášť užitočné, keď klient potrebuje zbierať informácie z viacerých zdrojov.
6. **Choreography (Event-Based Communication):**
   * Namiesto použitia orchestrácie, kde centrálny koordinátor riadi interakcie medzi službami, choreografia používa udalosti na komunikáciu medzi službami, čo znižuje závislosti a zvyšuje odolnosť.
7. **Decomposition:**
   * Rozkladanie veľkých monolitických aplikácií na menšie, nezávisle nasaditeľné mikroslužby. Tento vzor pomáha zvýšiť flexibilitu a škálovateľnosť systému.
8. **Sidecar:**
   * Vzor Sidecar umožňuje pridať pomocné funkcie k mikroslužbe, ako je monitoring, logovanie alebo konfigurácia, bez zmeny samotnej mikroslužby.

Tieto a ďalšie vzory v architektúre mikroslužieb pomáhajú riešiť bežné výzvy, zvyšujú modularitu, odolnosť a škálovateľnosť systému a umožňujú nezávislý vývoj a nasadzovanie jednotlivých služieb.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako napísať/navrhnúť dobre testovateľný kód (triedu)? Ako navrhnúť štruktúru? Ako spraviť závislosti?**

Na písanie dobre testovateľného kódu je dôležité dodržiavať niekoľko základných princípov a postupov, ktoré zjednodušia testovanie jednotiek a zvýšia kvalitu kódu. Tu sú kľúčové stratégie, ktoré by ste mali zvážiť pri návrhu a implementácii tried:

1. **Čistá architektúra:**
   * Rozdeľte kód na logické vrstvy (napríklad prezentáciu, logiku, prístup k dátam), kde každá vrstva má jasne definované úlohy a zodpovednosti. To umožňuje testovať každú vrstvu nezávisle.
2. **Single Responsibility Principle (SRP):**
   * Každá trieda by mala mať len jednu zodpovednosť alebo dôvod na zmenu. Toto zjednodušuje testovanie, pretože sa môžete zamerať na jednu funkcionalitu alebo správanie v rámci jednotkových testov.
3. **Dependency Injection (DI):**
   * Použitie DI umožňuje oddeliť závislosti triedy od jej implementácie. Tým pádom môžete ľahko nahradzovať závislosti mock objektmi alebo stubmi počas testovania, čo uľahčuje izolované testovanie jednotlivých komponentov.
4. **Interface Segregation Principle (ISP):**
   * Definujte úzke rozhrania, ktoré sú špecifické pre klientov, ktorí ich používajú. To umožňuje jednoduchšie mockovanie a testovanie, pretože testy môžu implementovať alebo mockovať len relevantné metódy.
5. **Avoid Static Methods:**
   * Statické metódy sú náročné na testovanie, pretože ich nie je možné preťažiť alebo nahrať mock objektmi. Kde je to možné, dajte prednosť inštancným metódam.
6. **Test-Driven Development (TDD):**
   * Píšte testy predtým, ako implementujete funkcionality. TDD vás núti premýšľať o tom, ako navrhnúť kód tak, aby bol ľahko testovateľný, a zároveň zabezpečuje, že máte vysoké pokrytie testami od začiatku vývojového cyklu.
7. **Refaktorizácia a testovanie existujúceho kódu:**
   * Pri refaktorizácii existujúceho kódu neustále zlepšujte jeho testovateľnosť. Rozdeľujte veľké metódy na menšie, oddeliteľné časti a zabezpečte, aby boli závislosti správne injektované.
8. **Používanie Mocking Frameworks:**
   * Používajte nástroje ako Mockito alebo EasyMock na vytváranie mock objektov pre závislosti. Týmto spôsobom môžete simulovať správanie závislostí bez potreby ich skutočnej implementácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké sú najbežnejšie zraniteľnosti webových aplikácií?**

Najbežnejšie zraniteľnosti webových aplikácií, podľa organizácie OWASP (Open Web Application Security Project), ktorá vydáva pravidelné správy o top zraniteľnostiach, zahŕňajú nasledujúce:

1. **Injection (Injektáž):**
   * SQL Injection, Command Injection, LDAP Injection, a ďalšie typy injektáže umožňujú útočníkom vkladať zlomyseľný kód do aplikácie, ktorý môže byť potom vykonávaný systémom alebo databázou.
2. **Broken Authentication (Porušená autentifikácia):**
   * Nedostatočné zabezpečenie procesov autentifikácie a správy relácií, ktoré môžu umožniť útočníkom prevziať kontrolu nad účtami používateľov.
3. **Sensitive Data Exposure (Expozícia citlivých dát):**
   * Nedostatočné zabezpečenie pri prenose alebo ukladaní citlivých dát, ako sú heslá, kreditné karty alebo osobné údaje, môže viesť k ich odcudzeniu alebo zneužitiu.
4. **XML External Entities (XXE):**
   * Zraniteľnosti spojené s procesovaním XML, ktoré môžu umožniť útočníkom čítať súbory na serveri, vykonávať vzdialené požiadavky alebo dokonca spúšťať zlomyseľný kód.
5. **Broken Access Control (Porušená kontrola prístupu):**
   * Nedostatočné overenie oprávnení používateľov môže umožniť neoprávnený prístup k dátam alebo funkcionalitám.
6. **Security Misconfiguration (Zlá konfigurácia zabezpečenia):**
   * Chyby v konfigurácii aplikácie, databázy, servera alebo platformy, ktoré môžu vytvoriť bezpečnostné medzery.
7. **Cross-Site Scripting (XSS):**
   * Zraniteľnosť umožňujúca útočníkom vkladať zlomyseľné skripty do webových stránok, ktoré sú potom vykonávané v prehliadači obete.
8. **Insecure Deserialization (Nedôveryhodná deserializácia):**
   * Nedostatočné overenie alebo kontrola serializovaných objektov môže umožniť útočníkom vykonávať zlomyseľné operácie.
9. **Using Components with Known Vulnerabilities (Používanie komponentov so známymi zraniteľnosťami):**
   * Závislosti na knižniciach, frameworkoch alebo iných softvérových komponentoch, ktoré majú známe zraniteľnosti, bez ich aktualizácie alebo zabezpečenia.
10. **Insufficient Logging & Monitoring (Nedostatočné logovanie a monitorovanie):**
    * Nedostatočné logovanie a monitorovanie, ktoré bránia včasnému odhaleniu alebo reagovaniu na bezpečnostné incidenty.

Rozpoznanie a náležité riešenie týchto zraniteľností je kľúčové pre zabezpečenie webových aplikácií a ochranu citlivých dát.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Na čo je dobrý INDEX v SQL? Na čo je dobré poradie stĺpcov (pri indexovaní)?**  
Indexy v SQL sú nástroje, ktoré zlepšujú rýchlosť vykonávania dotazov na databáze tým, že efektívne organizujú dáta. Podobne ako index v knihe, ktorý vám umožňuje rýchlo nájsť požadované informácie bez nutnosti prečítať celú knihu, index v databáze umožňuje databázovému systému rýchlejšie nájsť a načítať dáta bez nutnosti prehľadávať celú tabuľku.

1. **Výhody indexu:**
   * **Zvýšenie rýchlosti vyhľadávania:** Indexy značne urýchľujú vyhľadávanie a získavanie dát, najmä v tabuľkách s veľkým množstvom riadkov.
   * **Optimalizácia spojení (JOINs):** Indexy sú kriticky dôležité pre rýchle vykonávanie operácií JOIN, keďže umožňujú databázovému systému rýchlejšie nájsť prepojené dáta medzi tabuľkami.
   * **Zlepšenie výkonu pri zoradení a agregácii:** Indexy môžu zrýchliť operácie, ktoré vyžadujú zoradenie (ORDER BY) alebo agregáciu (GROUP BY) dát.
2. **Poradie stĺpcov pri indexovaní:**
   * **Optimalizácia pre konkrétne dotazy:** Poradie stĺpcov v indexe je dôležité, pretože databázové systémy využívajú index zľava doprava. Ak máte index definovaný na viacerých stĺpcoch, bude najefektívnejší pre dotazy, ktoré využívajú predné stĺpce indexu.
   * **Predchádzanie vytváraniu nadbytočných indexov:** Inteligentným určením poradia stĺpcov môžete zabezpečiť, že index je využiteľný pre rôzne dotazy, čo znižuje potrebu vytvárať mnoho rôznych indexov.
3. **Ako určiť poradie stĺpcov:**
   * **Analýza najčastejšie používaných dotazov:** Poradie by malo byť zvolené na základe toho, ktoré stĺpce sú najčastejšie používané v WHERE klauzuli alebo sú súčasťou JOIN podmienok.
   * **Cardinality:** Stĺpce s vyššou kardinalitou (väčším počtom jedinečných hodnôt) by mali byť umiestnené pred stĺpce s nižšou kardinalitou.

Vytváranie indexu nie je vždy bez dôsledkov – zatiaľ čo čítanie dát sa urýchli, zápis (INSERT, UPDATE, DELETE operácie) môže byť spomalený, pretože databázový systém musí aktualizovať indexy. Z tohto dôvodu je dôležité indexy starostlivo plánovať a používať ich tam, kde prinášajú najväčší prínos.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Záleží na poradí stĺpcov pokiaľ je index multistĺpcový? Prečo?**  
Áno, poradie stĺpcov v multistĺpcovom indexe má veľký význam, pretože ovplyvňuje efektívnosť indexu pri vyhľadávaní dát. Tu sú hlavné dôvody, prečo je poradie stĺpcov dôležité:

1. **Využitie indexu podľa dotazov:**
   * SQL databázy typicky využívajú indexy zľava doprava. To znamená, že ak máte index na stĺpcoch (A, B, C) a vykonáte dotaz, ktorý filtruje podľa stĺpca B, index nemusí byť použitý efektívne alebo vôbec, ak dotaz neobsahuje aj stĺpec A. Na druhej strane, ak dotaz filtruje podľa stĺpcov A a B (v tomto poradí), index bude veľmi účinný.
2. **Selektivita stĺpcov:**
   * Stĺpce s vyššou selektivitou (väčší počet jedinečných hodnôt) by mali byť v indexe umiestnené skôr, pretože vytvárajú viac granulárny index a umožňujú databáze efektívnejšie zúžiť vyhľadávanie. Ak je stĺpec s vysokou selektivitou umiestnený prvý, index dokáže účinne rozdeliť dáta a zrýchliť vyhľadávanie.
3. **Pokrytie dotazov:**
   * Multistĺpcové indexy môžu byť tiež navrhnuté tak, aby pokryli konkrétne dotazy. To znamená, že dotaz môže získať všetky potrebné dáta priamo z indexu bez potreby pristupovať k samotnej tabuľke. Poradie stĺpcov v indexe by malo byť zvolené tak, aby maximalizovalo počet dotazov, ktoré môžu byť pokryté.
4. **Indexové skenovanie a porovnávanie rozsahov:**
   * Pri vykonávaní operácií, ktoré zahŕňajú rozsahové dotazy (napr. **BETWEEN**, **>**, **<**), je dôležité, aby stĺpec používaný v týchto operáciách bol v indexe umiestnený čo najskôr. Ak sa rozsahové dotazy vykonávajú na stĺpci, ktorý nie je na začiatku indexu, efektívnosť indexu môže byť výrazne znížená.

Vzhľadom na tieto faktory je pri návrhu databázových indexov dôležité zvážiť nielen, ktoré stĺpce indexovať, ale aj ich poradie v indexe, aby ste maximalizovali výkon a efektívnosť vašich databázových operácií.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako ste pristupoval k databáze v SPRING-u?**

Pri práci s databázami v Spring frameworku existuje niekoľko prístupov, ktoré môžete použiť na interakciu s databázami. Tu je prehľad niektorých z najbežnejších spôsobov:

1. **JdbcTemplate:**
   * **JdbcTemplate** je klasický prístup Springu k práci s databázami, ktorý zjednodušuje kód potrebný na vykonávanie dotazov, aktualizácií a čítanie výsledkov.
   * Pomocou **JdbcTemplate** môžete vykonávať SQL operácie, spracovať výsledky a zachytávať výnimky v konzistentnom štýle.
   * Je ideálny pre situácie, keď potrebujete priamy kontrolu nad SQL dotazmi.
2. **Spring Data JPA:**
   * Spring Data JPA zjednodušuje prácu s dátami v databázach prostredníctvom JPA tým, že poskytuje vyššiu úroveň abstrakcie nad JPA. Umožňuje vám definovať repository rozhrania, na ktorých Spring automaticky vytvára implementácie za behu.
   * Môžete jednoducho definovať metódy v repository rozhraniach, a Spring Data JPA bude generovať implementáciu dotazu na základe konvencie mena metódy.
3. **Spring Data JDBC:**
   * Ak nechcete používať plnú špecifikáciu JPA, Spring Data JDBC predstavuje jednoduchší prístup k mapovaniu relačných databázových operácií na objekty v Jave, s menším množstvom magic a viac explicitného správania.
   * Spring Data JDBC je vhodný pre aplikácie, kde je potrebná priama kontrola nad databázovými operáciami s jednoduchším mapovaním než v prípade JPA.
4. **Transaction Management:**
   * Spring poskytuje deklaratívne transakčné riadenie, ktoré môžete použiť na zabezpečenie konzistencie dát pri vykonávaní operácií, ktoré by mali byť realizované ako atómové jednotky.
   * Môžete anotovať služby alebo metódy anotáciou **@Transactional** a Spring sa postará o riadenie transakcií.
5. **Spring Data MongoDB, Spring Data Redis, atď.:**
   * Pre NoSQL databázy, Spring poskytuje projekt Spring Data, ktorý obsahuje moduly pre rôzne NoSQL databázy, ako sú MongoDB, Redis, Cassandra, atď.
   * Tieto moduly uľahčujú prácu s NoSQL databázami pomocou konzistentného programátorského modelu, podobne ako Spring Data JPA pre relačné databázy.

Pri výbere prístupu by ste mali zvážiť vaše požiadavky na aplikáciu, úroveň abstrakcie, ktorú potrebujete, a konkrétne charakteristiky a vlastnosti databázy, s ktorou pracujete.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Riešil ste niekedy MEMORY LEAKS alebo DEAD LOCK?**

Áno, v rámci môjho vývoja a diagnostiky softvérových aplikácií som sa stretával s riešením problémov spojených s memory leaks (únikmi pamäte) a deadlocks (vzájomnými zablokovaniami).

Memory Leaks

Memory leak vzniká, keď aplikácia dynamicky alokuje pamäť na heap-e a neuvolní ju správne po tom, čo už nie je potrebná. To môže viesť k postupnému vyčerpaniu dostupnej pamäte, čo spôsobuje pomalý chod aplikácie alebo dokonca jej kolaps.

**Riešenie Memory Leaks:**

* **Profiling a Monitoring:** Použitie nástrojov na profiling a monitoring aplikácie, ako sú VisualVM, JProfiler, alebo Eclipse Memory Analyzer Tool (MAT), na identifikáciu objektov, ktoré zaberajú neúmerne veľa pamäte a neuvolňujú sa.
* **Kódová Analýza:** Detailná analýza kódu na identifikáciu miest, kde objekty nie sú správne uvolňované. To môže zahŕňať nevhodné použitie statických premenných, kolekcií, listenerov, atď.
* **Refaktoring:** Upravenie kódu na odstránenie úniku pamäte, napríklad použitím slabých referencií (WeakReferences), správnym riadením životného cyklu objektov, alebo zlepšením logiky uvolňovania zdrojov.

Deadlocks

Deadlock nastáva, keď dve alebo viac vlákien navzájom čakajú na zdroje, ktoré drží to druhé, čo vedie k tomu, že žiadne z nich nemôže pokračovať vo svojej činnosti. To je často výsledkom nevhodného riadenia synchronizácie a zamknutia zdrojov.

**Riešenie Deadlocks:**

* **Analýza Thread Dumpov:** Použitie thread dumpov na identifikáciu zamknutých vlákien a zdrojov, ktoré držia. Nástroje ako jstack alebo IDE debuggery môžu poskytnúť cenné informácie o stave vlákien a zamknutí.
* **Prehodnotenie Synchronizácie:** Preskúmanie a refaktoring kódu na minimalizáciu potreby zamknutia alebo na použitie pokročilejších konštruktov pre konkurenciu z Java Concurrent package, ako sú ReentrantLocks, Semaphore, atď., ktoré poskytujú viac kontroly nad zamknutím.
* **Vyhnúť sa Vzorom, Ktoré Vedú k Deadlocku:** Napríklad, vždy získavajte zamknutia v konzistentnom poradí a uvoľňujte ich v opačnom poradí, používajte timeouty pri pokusoch o získanie zamknutia, a podobne.

Riešenie týchto problémov vyžaduje hĺbkové pochopenie fungovania Java Virtual Machine (JVM), správne používanie nástrojov na diagnostiku a schopnosť analyzovať a refaktorovať kód tak, aby sa predišlo budúcim problémom.

Top of Form

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako by ste odhalil DEAD LOCK?**  
Áno, v rámci môjho vývoja a diagnostiky softvérových aplikácií som sa stretával s riešením problémov spojených s memory leaks (únikmi pamäte) a deadlocks (vzájomnými zablokovaniami).

Memory Leaks

Memory leak vzniká, keď aplikácia dynamicky alokuje pamäť na heap-e a neuvolní ju správne po tom, čo už nie je potrebná. To môže viesť k postupnému vyčerpaniu dostupnej pamäte, čo spôsobuje pomalý chod aplikácie alebo dokonca jej kolaps.

**Riešenie Memory Leaks:**

* **Profiling a Monitoring:** Použitie nástrojov na profiling a monitoring aplikácie, ako sú VisualVM, JProfiler, alebo Eclipse Memory Analyzer Tool (MAT), na identifikáciu objektov, ktoré zaberajú neúmerne veľa pamäte a neuvolňujú sa.
* **Kódová Analýza:** Detailná analýza kódu na identifikáciu miest, kde objekty nie sú správne uvolňované. To môže zahŕňať nevhodné použitie statických premenných, kolekcií, listenerov, atď.
* **Refaktoring:** Upravenie kódu na odstránenie úniku pamäte, napríklad použitím slabých referencií (WeakReferences), správnym riadením životného cyklu objektov, alebo zlepšením logiky uvolňovania zdrojov.

Deadlocks

Deadlock nastáva, keď dve alebo viac vlákien navzájom čakajú na zdroje, ktoré drží to druhé, čo vedie k tomu, že žiadne z nich nemôže pokračovať vo svojej činnosti. To je často výsledkom nevhodného riadenia synchronizácie a zamknutia zdrojov.

**Riešenie Deadlocks:**

* **Analýza Thread Dumpov:** Použitie thread dumpov na identifikáciu zamknutých vlákien a zdrojov, ktoré držia. Nástroje ako jstack alebo IDE debuggery môžu poskytnúť cenné informácie o stave vlákien a zamknutí.
* **Prehodnotenie Synchronizácie:** Preskúmanie a refaktoring kódu na minimalizáciu potreby zamknutia alebo na použitie pokročilejších konštruktov pre konkurenciu z Java Concurrent package, ako sú ReentrantLocks, Semaphore, atď., ktoré poskytujú viac kontroly nad zamknutím.
* **Vyhnúť sa Vzorom, Ktoré Vedú k Deadlocku:** Napríklad, vždy získavajte zamknutia v konzistentnom poradí a uvoľňujte ich v opačnom poradí, používajte timeouty pri pokusoch o získanie zamknutia, a podobne.

Riešenie týchto problémov vyžaduje hĺbkové pochopenie fungovania Java Virtual Machine (JVM), správne používanie nástrojov na diagnostiku a schopnosť analyzovať a refaktorovať kód tak, aby sa predišlo budúcim problémom.Top of Form

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Hovorí Vám niečo THREAT DUMP?**

"thread dump" (v slovenčine niekedy nazývaný aj výpis vlákien) je dobre známy v kontexte vývoja softvéru a ladění výkonu aplikácií. Thread dump je snímka, ktorá ukazuje, čo všetky vlákna v Jave (alebo inom programovacom jazyku) robia v danom momente. Poskytuje veľmi detailný pohľad na stavy vlákien, zásobník volaní pre každé vlákno a iné informácie, ktoré môžu pomôcť identifikovať a vyriešiť problémy ako sú deadlocky, zbytočné čakanie na zdroje, nadmerné využívanie CPU a podobne.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako ste na tom s DESIGN PATTERNAMI?**  
Design patterny sú osvedčené riešenia pre často sa vyskytujúce problémy v softvérovom dizajne. Využívajú sa na zlepšenie čitateľnosti, údržby kódu a jeho opätovnej použiteľnosti. V kontexte objektovo orientovaného programovania (OOP) a princípov SOLID, tieto vzory pomáhajú dosiahnuť čistý a efektívny dizajn softvéru. Tu je prehľad základných kategórií design patternov:

1. **Vytváracie vzory (Creational patterns):** Pomáhajú pri vytváraní objektov, pričom skrývajú logiku ich vytvárania a umožňujú objektom byť vytvoreným spôsobom vhodným pre danú situáciu.
   * **Singleton:** Zaručuje, že trieda má len jednu inštanciu, a poskytuje globálny prístupový bod k tejto inštancii.
   * **Factory Method:** Definuje rozhranie pre vytváranie objektu, ale necháva podtriedy rozhodnúť, ktorá trieda sa má inštancovať.
   * **Abstract Factory:** Poskytuje rozhranie pre vytváranie rodín súvisiacich alebo závislých objektov bez špecifikovania ich konkrétnych tried.
   * **Builder:** Oddeluje konštrukciu komplexného objektu od jeho reprezentácie, takže ten istý konštrukčný proces môže vytvoriť rôzne reprezentácie.
   * **Prototype:** Umožňuje kopírovanie existujúcich objektov bez potreby ich znovu vytvárania a bez závislosti na ich triedach.
2. **Štrukturálne vzory (Structural patterns):** Riešia spôsob, akým sú triedy a objekty komponované do väčších štruktúr.
   * **Adapter (Wrapper):** Umožňuje objektom s nekompatibilnými rozhraniami pracovať spoločne.
   * **Composite:** Komponuje objekty do stromových štruktúr, aby reprezentovali hierarchiu časť-celok.
   * **Proxy:** Poskytuje náhradníka alebo zástupcu pre iný objekt, aby sa ovládal prístup k nemu.
   * **Flyweight:** Umožňuje použitie veľkého množstva objektov zdieľaním čo najväčšieho množstva kódu/stavu medzi nimi.
   * **Bridge:** Oddeluje abstrakciu od jej implementácie, takže obe sa môžu meniť nezávisle.
3. **Behaviorálne vzory (Behavioral patterns):** Riešia efektívnu komunikáciu medzi objektmi.
   * **Observer:** Umožňuje objektu publikovať zmeny svojho stavu a iným objektom sa prihlásiť k odberu týchto zmien.
   * **Strategy:** Umožňuje definovať rodinu algoritmov, zapuzdriť každý z nich a urobiť ich zamieňateľnými.
   * **Command:** Zapuzdruje požiadavku ako objekt, čím umožňuje parametrizáciu klientov s rôznymi požiadavkami, frontami alebo operáciami a podporuje reverzibilné operácie.
   * **State:** Umožňuje objektu meniť svoje správanie, keď sa zmení jeho vnútorný stav. Objekt sa zdá byť meniaci svoju triedu.
   * **Template Method:** Definuje kostru algoritmu v metóde, odložiac niektoré kroky na podtriedy. Umožňuje podtriedam predefinovať určité kroky algoritmu bez zmeny jeho štruktúry.

Tieto vzory sú základné stavebné kamene pre návrh flexibilného a udržateľného softvéru v jazyku Java, ale aj v mnohých iných objektovo orientovaných jazykoch. Implementácia týchto vzorov s ohľadom na princípy SOLID môže výrazne zlepšiť kvalitu a udržateľnosť vášho kódu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aké DESIGN PATTERNS poznáte (okrem Factory a Singleton)?**

Vytváracie vzory (Creational patterns)

1. **Singleton**
   * **Popis:** Zabezpečuje, že trieda má len jednu inštanciu a poskytuje k nej globálny prístupový bod.
   * **Použitie:** Používa sa, keď je potrebné mať presne jednu inštanciu triedy cez celú aplikáciu, ako je napríklad konfiguračný manažér alebo správca pripojení k databáze.
   * **Príklad:** Implementácia konfiguračného manažéra, ktorý načítava konfiguráciu pri štarte a poskytuje univerzálny prístup k nastaveniam.
2. **Factory Method**
   * **Popis:** Definuje rozhranie pre vytváranie objektu, ale necháva podtriedy, aby rozhodli o tom, ktorá trieda sa má inštancovať.
   * **Použitie:** Vhodný, keď existuje viacero možných tried, ktoré môžu byť vytvorené, ale rozhodnutie o tom, ktorú triedu použiť, závisí od vstupných dát alebo konfigurácie.
   * **Príklad:** Vytvorenie rôznych typov dokumentov v textovom editore, kde každý typ dokumentu je vytvorený pomocou špecifického factory method.
3. **Abstract Factory**
   * **Popis:** Poskytuje rozhranie pre vytváranie rodín súvisiacich alebo závislých objektov bez špecifikácie ich konkrétnych tried.
   * **Použitie:** Používa sa, keď systém má byť nezávislý od toho, ako sú jeho produkty vytvárané, komponované a reprezentované.
   * **Príklad:** Vytváranie užívateľského rozhrania, kde každý prvok (tlačidlá, okná, ikony) môže mať rôzne štýly závislé od operačného systému alebo témy bez zmeny kódu, ktorý tieto prvky používa.
4. **Builder**
   * **Popis:** Oddeluje konštrukciu komplexného objektu od jeho reprezentácie, takže ten istý konštrukčný proces môže vytvoriť rôzne reprezentácie.
   * **Použitie:** Ideálny pre vytváranie komplexných objektov s mnohými voliteľnými parametrami alebo keď je proces tvorby objektu potrebné rozdeliť na viacero krokov.
   * **Príklad:** Vytváranie rôznych typov pizze s rôznymi ingredienciami a prísadami, kde každý typ pizze vyžaduje iný postup a kombináciu ingrediencií.
5. **Prototype**
   * **Popis:** Umožňuje kopírovanie existujúcich objektov bez potreby znovu ich vytvárania a bez závislosti na ich triedach.
   * **Použitie:** Vhodné, keď je vytvorenie inštancie drahé alebo komplexné, ale potrebujete podobný alebo upravený objekt.
   * **Príklad:** Duplikovanie grafických objektov v grafickom editore, kde každý objekt môže mať množstvo nastavení, ktoré chceme zachovať pri jeho kopírovaní.

Štrukturálne vzory (Structural patterns)

1. **Adapter (Wrapper)**
   * **Popis:** Umožňuje objektom s nekompatibilnými rozhraniami pracovať spoločne.
   * **Použitie:** Používa sa, keď chcete, aby triedy s nekompatibilnými rozhraniami mohli spolupracovať.
   * **Príklad:** Integrovanie externých knižníc alebo API, ktoré majú odlišné rozhrania, do vašej aplikácie bez zmeny ich kódu.
2. **Composite**
   * **Popis:** Umožňuje skupinám objektov byť ošetreným rovnako ako jednotlivým objektom.
   * **Použitie:** Vhodný pre prácu s hierarchickými štruktúrami, kde objekty môžu obsahovať iné objekty, ktoré môžu obsahovať ďalšie objekty atď., vytvárajúc stromovú štruktúru.
   * **Príklad:** Grafické užívateľské rozhrania, kde môžete mať komponenty ako sú okná, ktoré obsahujú panely, ktoré ďalej obsahujú tlačidlá, checkboxy atď.
3. **Proxy**
   * **Popis:** Poskytuje náhradníka alebo zástupcu pre iný objekt, aby sa mohli kontrolovať prístupy k nemu.
   * **Použitie:** Užitočné pre zabezpečenie, manažment zdrojov, alebo pre lazy inicializáciu objektov, ktorých vytvorenie je náročné na zdroje.
   * **Príklad:** Zastupovanie sieťových požiadaviek alebo operácií s databázou, kde Proxy môže pridávať dodatočné funkcie ako caching alebo správu transakcií.
4. **Flyweight**
   * **Popis:** Umožňuje zdieľať veľké množstvo objektov s minimálnymi nákladmi na pamäť.
   * **Použitie:** Vhodný, keď sa v aplikácii vyskytuje veľké množstvo objektov s podobným stavom, ktoré môžete externalizovať a zdieľať.
   * **Príklad:** Textový editor, ktorý ukladá iba jednu inštanciu každého znaku, bez ohľadu na to, koľkokrát sa v dokumente vyskytuje, čím šetrí pamäť.
5. **Facade**
   * **Popis:** Poskytuje jednoduché rozhranie k zložitému systému subsystémov, čím zjednodušuje komunikáciu s ním.
   * **Použitie:** Používa sa na poskytnutie jednoduchého rozhrania k zložitému alebo štruktúrovanému systému.
   * **Príklad:** Systém domácej automatizácie, kde Fasáda poskytuje jednoduché rozhranie pre ovládanie rôznych zariadení v dome, ako sú svetlá, teplota, bezpečnostné systémy atď.
6. **Bridge**
   * **Popis:** Oddeluje abstrakciu od implementácie, takže obe sa môžu meniť nezávisle.
   * **Použitie:** Umožňuje, aby sa implementácia a abstrakcia rozvíjali nezávisle. Vhodné pre systémy, kde je potrebná flexibilita vo voľbe alebo zmenách implementácie.
   * **Príklad:** Grafické aplikácie, kde môžete mať abstrakciu tvaru (napr. kruh, štvorec) a niekoľko implementácií pre kreslenie (napr. kreslenie pomocou rôznych grafických knižníc).

Behaviorálne vzory (Behavioral patterns)

1. **Observer**
   * **Popis:** Umožňuje objektu publikovať zmeny svojho stavu a iným objektom, tzv. pozorovateľom, sa na tieto zmeny prihlásiť a reagovať.
   * **Použitie:** Vhodné pre situácie, kde zmena v jednom objekte vyžaduje zmeny v jednom alebo viacerých iných objektoch, a nechcete, aby boli objekty pevne spájané.
   * **Príklad:** Správa udalostí v užívateľských rozhraniach, kde zmeny v užívateľskom vstupe (napr. kliknutie na tlačidlo) vyvolajú reakcie v iných častiach aplikácie.
2. **Strategy**
   * **Popis:** Umožňuje definovať rodinu algoritmov, zapuzdriť každý z nich a urobiť ich vzájomne vymeniteľné. Stratégia umožňuje algoritmu meniť sa nezávisle od klientov, ktorí ho používajú.
   * **Použitie:** Používa sa, keď sú v aplikácii potrebné rôzne varianty nejakého algoritmu alebo keď je algoritmus pravdepodobne predmetom zmeny v budúcnosti.
   * **Príklad:** Rôzne spôsoby zoradenia kolekcie objektov, kde každý spôsob zoradenia je implementovaný ako stratégia, a môže byť dynamicky zmenený podľa potreby.
3. **Command**
   * **Popis:** Umožňuje zapúzdriť požiadavku ako objekt, čím sa zjednodušuje parametrizácia a správa metód volaní, front požiadaviek, alebo operácií vykonávaných v transakciách.
   * **Použitie:** Vhodný pre implementáciu funkcií ako sú undo/redo, callbacky a operácie vo frontách.
   * **Príklad:** Grafický editor, kde každá akcia, ako je kreslenie tvaru alebo zmazanie, je reprezentovaná ako objekt príkazu, ktorý môže byť vykonaný, vrátený späť alebo zopakovaný.
4. **State**
   * **Popis:** Umožňuje objektu meniť svoje správanie, keď sa zmení jeho interný stav. Objekt sa bude zdať, že mení svoju triedu.
   * **Použitie:** Používa sa, keď objekty majú rôzne stavy s rôznymi správaniami a stav objektu musí byť zmenený za behu.
   * **Príklad:** Textový editor, ktorý mení správanie kurzora alebo interakcií v závislosti od toho, či je v režime vkladania alebo v režime výberu.
5. **Chain of Responsibility**
   * **Popis:** Umožňuje prenášať požiadavky pozdĺž reťazca spracovateľov. Pri prechode každý spracovateľ rozhoduje, či bude požiadavku spracovať, alebo ju prepošle ďalej v reťazci.
   * **Použitie:** Ideálne, keď máte viac ako jeden objekt, ktorý môže spracovať určitú požiadavku, a chcete určiť spracovateľa za behu.
   * **Príklad:** Middleware v webových aplikáciách, kde každá vrstva rozhoduje, či bude požiadavku spracovať alebo ju prepošle ďalej.
6. **Mediator**
   * **Popis:** Umožňuje znížiť zložitosť komunikácie medzi viacerými objektmi alebo triedami tým, že zabezpečuje, že namiesto toho, aby komunikovali priamo, budú komunikovať prostredníctvom mediátora.
   * **Použitie:** Používa sa na zjednodušenie komunikácie medzi viacerými objektmi alebo systémovými komponentami.
   * **Príklad:** Kontrolný systém pre leteckú dopravu, kde mediátor koordinuje všetky komunikácie medzi lietadlami a letovými kontrolami.
7. **Visitor**
   * **Popis:** Umožňuje pridať nové operácie do tried bez zmeny týchto tried. Operácie sú vykonávané na objektoch rôznych tried.
   * **Použitie:** Užitočný, keď máte komplexnú objektovú štruktúru a chcete na nej vykonať operácie, ktoré nechcete začleňovať do tried objektov.
   * **Príklad:** XML alebo HTML dokument, kde visitor môže byť použitý na vykonanie operácií, ako je vykreslenie alebo extrakcia dát, na rôzne typy uzlov bez zmeny tried uzlov.
8. **Memento**
   * **Popis:** Umožňuje uložiť a neskôr obnoviť predchádzajúci stav objektu bez odhaľovania podrobností o jeho implementácii.
   * **Použitie:** Používa sa pre implementáciu funkcionalít ako undo alebo snapshot.
   * **Príklad:** Editor textu, kde môžete uložiť stav dokumentu a neskôr ho obnoviť, ak je to potrebné.
9. **Interpreter**
   * **Popis:** Určený na poskytnutie spôsobu, ako vyhodnocovať gramatiku alebo výrazy jazyka v určenom jazyku.
   * **Použitie:** Užitočný pri implementácii špecifických alebo doménovo špecifických jazykov.
   * **Príklad:** Vyhodnotenie matematických výrazov alebo SQL dotazov.
10. **Template Method**
    * **Popis:** Definuje kostru algoritmu v metóde, odložiac niektoré kroky na podtriedy. Umožňuje podtriedam predefinovať určité kroky algoritmu bez zmeny jeho štruktúry.
    * **Použitie:** Ideálne, keď majú viaceré triedy podobné algoritmy s miernymi variáciami v niektorých krokoch.
    * **Príklad:** Rôzne recepty na prípravu čaju alebo kávy, kde kroky sú podobné, ale niektoré špecifiké akcie sa líšia v závislosti od nápoja.

Tieto vzory pokrývajú základné princípy návrhu softvéru a pomáhajú vytvárať flexibilné, udržateľné a rozširiteľné softvérové aplikácie. Každý vzor má svoje špecifické použitie a je dôležité pochopiť, kedy a ako každý z nich použiť v kontexte vašich projektov.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Poznáte SAGA pattern?**  
Áno, SAGA pattern je návrhový vzor používaný v mikroslužbách a distribuovaných systémoch na správu transakcií, ktoré vyžadujú koordináciu medzi viacerými službami. Cieľom tohto vzoru je zabezpečiť konzistentnosť a integritu dát naprieč službami v prípade, že nie je možné použiť tradičné databázové transakcie (napríklad kvôli ich distribuovanej povahy). SAGA pattern rozdeľuje tieto dlhotrvajúce transakcie na sériu menších transakcií, pričom každá z nich môže byť potvrdená nezávisle.

Princíp fungovania SAGA patternu

SAGA pattern môže byť implementovaný dvoma hlavnými spôsobmi: ako sekvenčný tok (Orchestration) alebo ako decentralizovaný tok (Choreography).

1. **Orchestration (Orchestrácia):**
   * Pri orchestrovanej sage je definovaný centrálny koordinátor (orchestrátor), ktorý riadi celý proces, rozhoduje o tom, aké kroky sa majú vykonať ďalej, a spracováva komunikáciu medzi mikroslužbami.
   * Orchestrátor zodpovedá za spustenie každej transakcie v sage a za spracovanie akýchkoľvek kompenzačných transakcií, ak je to potrebné (napríklad ak jedna z transakcií zlyhá).
2. **Choreography (Choreografia):**
   * Pri choreografii každá služba nezávisle vysiela a počúva udalosti od ostatných služieb. Týmto spôsobom môže reagovať na zmeny bez potreby centrálného koordinátora.
   * Služby vykonávajú svoje transakcie a v prípade úspechu alebo zlyhania vysielajú udalosti, ktoré informujú ostatné služby o stave transakcie, čo umožňuje systému ako celku pokračovať alebo vykonať kompenzačné akcie.

Príklad použitia

Predstavte si proces objednávky v e-commerce systéme, ktorý zahŕňa služby ako správa inventára, spracovanie platieb a logistiku. Pri vytváraní objednávky musí systém zabezpečiť, že sú produkty na sklade, platba je spracovaná a objednávka je pripravená na odoslanie. Ak niektorý z týchto krokov zlyhá (napríklad platba nie je úspešná), SAGA umožňuje definovať kompenzačné akcie (napríklad vrátenie produktov späť do inventára), aby bol systém vrátený do konzistentného stavu.

SAGA pattern je teda užitočný pre zabezpečenie konzistencie a odolnosti v distribuovaných systémoch, kde tradičné metódy transakčnej kontroly nie sú možné alebo vhodné.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký typ CACHEu má HIBERNATE?**

Hibernate poskytuje dva hlavné typy cache, ktoré sú navrhnuté na zvýšenie výkonnosti aplikácií tým, že znižujú počet potrebných databázových volaní:

1. **Prvá úroveň cache (First-Level Cache):**
   * Prvá úroveň cache je zapnutá predvolene a existuje ako súčasť **Session** objektu v Hibernate. Táto cache ukladá entity, ktoré boli načítané alebo uložené v rámci aktuálnej session.
   * Keď aplikácia požiada o entitu, ktorá už je v prvej úrovni cache, Hibernate ju vráti priamo z cache bez potreby ďalšieho databázového volania.
   * Táto cache je viazaná na konkrétnu session a nezdieľa sa medzi rôznymi sessions. Keď session skončí, cache sa vyprázdni.
2. **Druhá úroveň cache (Second-Level Cache):**
   * Na rozdiel od prvej úrovne cache je druhá úroveň cache v Hibernate voliteľná a môže byť zdieľaná medzi viacerými sessions a dokonca aj aplikáciami, ak sú nakonfigurované na používanie tej istej cache.
   * Druhá úroveň cache môže ukladať dáta na úrovni entít, kolekcií, dotazov a prírodných ID a pomáha výrazne znížiť počet databázových dotazov.
   * Na používanie druhej úrovne cache v Hibernate musíte explicitne zapnúť a nakonfigurovať ju, a tiež určiť, ktoré entity alebo kolekcie majú byť cacheované.
   * Hibernate podporuje viacero poskytovateľov cache pre druhú úroveň, napríklad Ehcache, Infinispan alebo Hazelcast.

**Cache dotazov (Query Cache):**

* Hibernate ponúka aj možnosť cacheovania výsledkov dotazov. Táto funkcia je užitočná, keď chcete cacheovať výsledky často používaných dotazov.
* Keď je cache dotazov zapnutá, Hibernate uloží výsledky dotazov do cache a pri opakovanom vykonaní rovnakého dotazu môže výsledky načítať priamo z cache namiesto toho, aby vykonával dotaz na databáze znova.
* Cache dotazov funguje najlepšie, keď sú výsledky dotazu relatívne statické, pretože časté zmeny v dátach môžu znižovať efektívnosť cache.

Správne používanie cache v Hibernate môže výrazne zlepšiť výkonnosť aplikácie znižovaním počtu nevyhnutných databázových operácií. Avšak, je dôležité správne nakonfigurovať a používať cache, pretože nevhodné použitie môže viesť k problémom s konzistenciou dát alebo výkonnosťou.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Popíšte mi konkrétny pattern (napr. DDD)**

Domain-Driven Design (DDD) je prístup k softvérovému vývoju, ktorý sa zameriava na komplexné pochopenie domény (oblasti problému) a modelovanie softvéru tak, aby odrážal jeho štruktúru a dynamiku. Cieľom DDD je vytvoriť softvér, ktorý je hlboko prepojený s odbornými znalosťami a logikou daného obchodného sektora, čím sa zvyšuje efektivita a znížuje komplexita systému. DDD je osobitne užitočný v projektoch, kde je obchodná logika komplexná a dynamicky sa mení.

Kľúčové koncepty DDD:

* **Ubiquitous Language (Všadeprítomný jazyk):** Jazyk, ktorý je zrozumiteľný pre odborníkov aj vývojárov, a používa sa na popis všetkých aspektov domény. Cieľom je zabrániť nedorozumeniam a zjednodušiť komunikáciu.
* **Bounded Context (Ohraničený kontext):** Definuje hranice a kontext, v rámci ktorých má doménový model platnosť. Umožňuje izolovať rôzne časti systému a zjednodušiť ich správu tým, že sa vyhnete zdieľaniu modelov medzi kontextmi, kde by to mohlo viesť k zámene alebo konfliktu.
* **Entities (Entity):** Objekty, ktoré majú svoju identitu, ktorá sa nelíši od ich atribútov. Identita umožňuje rozlišovať entity, aj keď ich atribúty sú identické.
* **Value Objects (Hodnotové objekty):** Objekty, ktoré opisujú vlastnosti alebo kvality niečoho, ale nemajú svoju identitu. Sú nemenné a často sa používajú na meranie alebo opis atribútov entity.
* **Aggregates:** Sú skupiny objektov (entity a hodnotové objekty), ktoré spolu fungujú ako jednotka pre účely údajovej zmeny. Každý agregát má koreňovú entitu, ktorá je zodpovedná za kontrolu prístupu k agregátu.
* **Repositories (Repositáre):** Používajú sa na oddelenie doménovej logiky od logiky ukladania údajov. Repositáre poskytujú abstrakciu, ktorá umožňuje manipuláciu s objektami domény bez potreby zaoberať sa detailami ich perzistencie.
* **Domain Events (Doménové udalosti):** Sú udalosti, ktoré vznikajú, keď sa v doméne niečo významné stane. Umožňujú komunikáciu medzi rôznymi časťami systému bez priamej závislosti medzi nimi.

Príklad použitia DDD:

Predstavte si vývoj softvéru pre správu zásob v maloobchodnom reťazci. Všadeprítomný jazyk by zahŕňal termíny ako "produkt", "inventár", "objednávka" atď. Bounded Context by mohol oddeliť správu zásob od e-commerce systému. Produkty by boli modelované ako entity s unikátnou identitou (napr. SKU kód). Hodnotové objekty by mohli zahŕňať cenu alebo rozmery produktu. Agregáty by mohli byť použité na skupinové spracovanie produktov a objednávok. Repositáre by uľahčili prácu s produktmi a objednávkami bez nutnosti riešiť, ako sú údaje uložené. Doménové udalosti by mohli signalizovať zmeny v inventári alebo nové objednávky.

DDD je teda mocný prístup, ktorý pomáha vytvárať vysoko adaptívne softvérové systémy s hlbokým zameraním na obchodné potreby a doménovú logiku.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aký je rozdiel medzi ENTITY a VALUE v patterne DDD?**

V Domain-Driven Design (DDD) sú pojmy "Entity" a "Value Object" dva základné stavebné bloky, ktoré sa používajú na modelovanie doménových konceptov. Rozlišovanie medzi týmito dvoma typmi objektov je kľúčové pre správne pochopenie a aplikáciu DDD prístupu. Rozdiely medzi Entity a Value Object môžeme charakterizovať nasledovne:

Entity

* **Identita:** Entity sú definované svojou identitou, ktorá sa zachováva počas celého životného cyklu objektu, aj keď sa jeho atribúty zmenia. Identita umožňuje odlíšiť jednu entitu od inej, dokonca aj v prípade, že ich ostatné vlastnosti sú identické.
* **Životný cyklus:** Entity majú výrazný životný cyklus. Môžu byť vytvorené, upravené alebo zničené a pri každej zmene sa zachováva ich identita.
* **Zameranie:** Zameriavajú sa na "čo" objekt je, t.j., ich identita je to, čo ich definuje a odlišuje od ostatných objektov.

Value Object

* **Nemennosť:** Value Objects sú nemenné; akonáhle sú vytvorené, ich stav sa nemôže zmeniť. Ak potrebujete "zmeniť" Value Object, vytvoríte novú inštanciu s novými hodnotami.
* **Nemajú identitu:** Na rozdiel od entít, Value Objects sú definované ich vlastnosťami a hodnotami, nie unikátnou identitou. Dve inštancie Value Object s rovnakými hodnotami sa považujú za ekvivalentné.
* **Zameranie:** Zameriavajú sa na "ako" niečo je reprezentované alebo na vlastnosti, ktoré opisujú objekt, a nie na samotnú identitu objektu.

Príklad

Predstavte si systém na správu knižnice:

* **Entity:** Knihy v knižnici by boli reprezentované ako entity. Každá kniha má jedinečné identifikačné číslo (napríklad ISBN), ktoré ju odlíši od iných kníh, aj keď majú rovnaký názov a autora. Identita knihy (jej ISBN) sa nezmení, aj keď sa kniha poškodí a musí byť opravená alebo zrestaurovaná.
* **Value Object:** Dátum vypožičania knihy by mohol byť reprezentovaný ako Value Object v rámci vypožičiavacieho systému. Dátum nemá svoju vlastnú identitu a je dôležitý len v kontexte toho, čo reprezentuje (t.j., kedy bola kniha vypožičaná). Ak zmeníte dátum vrátenia, jednoducho vytvoríte nový Value Object s novým dátumom.

Týmto rozdelením sa zjednodušuje modelovanie komplexných domén a zabezpečuje sa, že systém je navrhnutý s ohľadom na skutočnú podstatu a potreby domény.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Aky je to event driven design v porovnani s DDD?**

Event-Driven Design (EDD) a Domain-Driven Design (DDD) sú dve architektonické prístupy, ktoré sa môžu dopĺňať, ale majú odlišné zamerania a ciele pri návrhu softvérových systémov.

Event-Driven Design (EDD)

**Zameranie:** EDD sa sústreďuje na zachytávanie, generovanie, spracovanie a reagovanie na udalosti (events) v systéme. Udalosti sú signály, ktoré reprezentujú zmeny alebo dôležité akcie, ktoré sa udiali v systéme.

**Hlavné princípy:**

* **Decentralizácia:** Systém je navrhnutý okolo produkcie a konzumácie udalostí, čo umožňuje decentralizovanú architektúru.
* **Asynchrónnosť:** Komponenty systému komunikujú asynchrónne prostredníctvom udalostí, čo znižuje priamu závislosť medzi nimi a umožňuje vyššiu odolnosť a škálovateľnosť.
* **Reaktivita:** Systém je navrhnutý tak, aby aktívne reagoval na udalosti, čo umožňuje rýchle prispôsobenie sa zmenám a dynamické správanie.

Domain-Driven Design (DDD)

**Zameranie:** DDD je prístup k návrhu softvéru, ktorý sa zameriava na modelovanie softvéru tak, aby odrážal komplexné obchodné domény a ich logiku.

**Hlavné princípy:**

* **Ubiquitous Language:** Vývoj jasného a konzistentného doménového jazyka, ktorý je zrozumiteľný pre všetkých zainteresovaných (vývojárov, obchodných analytikov, atď.).
* **Bounded Contexts:** Rozdelenie systému na logicky oddelené časti, kde každá časť má svoj vlastný doménový model.
* **Model-Driven Design:** Vývoj a používanie bohatého doménového modelu, ktorý zjednodušuje komplexné obchodné procesy a pravidlá.

Porovnanie a Kombinácia

* **Zameranie:** Kým DDD sa zameriava na hĺbkové pochopenie a modelovanie obchodnej domény, EDD sa sústreďuje na dynamické spracovanie udalostí v systéme.
* **Komplementárnosť:** EDD a DDD sa môžu výborne dopĺňať. Napríklad, v rámci DDD môže byť použitý Event-Driven prístup na modelovanie a spracovanie doménových udalostí, čím sa zvyšuje reaktivita a flexibilita systému.
* **Integrácia:** V praxi je možné integrovať EDD do DDD prístupu tým, že doménové udalosti (Domain Events) sú použité na komunikáciu medzi bounded contexts alebo na reagovanie na zmeny v doméne asynchrónnym a decentralizovaným spôsobom.

Vo všeobecnosti, EDD môže poskytnúť efektívny mechanizmus pre implementáciu reaktívnych aspektov systému navrhnutého podľa princípov DDD, čo umožňuje lepšie zvládanie dynamických obchodných požiadaviek a zmeny v prostredí.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Ako sa rieši EXCEPTION HANDLING v SPRINGu?**

V Spring frameworku je exception handling (spracovanie výnimiek) široko podporované a umožňuje vývojárom efektívne spracovať chyby v aplikácii. Spring poskytuje rôzne mechanizmy pre spracovanie výnimiek, ktoré umožňujú centralizovanú manipuláciu s chybami, čím sa znižuje duplicita kódu a zvyšuje sa čitateľnosť. Tu sú kľúčové prístupy, ako Spring rieši exception handling:

1. @ExceptionHandler

**@ExceptionHandler** je anotácia používaná na metódach v rámci **@Controller** alebo **@RestController** tried, ktoré majú za úlohu spracovať výnimky vyvolané pri vykonávaní HTTP požiadaviek. Tento prístup umožňuje lokálne spracovanie výnimiek na úrovni kontroleru.

A screen shot of a computer code

Description automatically generated

2. @ControllerAdvice

**@ControllerAdvice** je anotácia, ktorá umožňuje globálne spracovanie výnimiek pre celú aplikáciu alebo pre špecifické kontrolery. Triedy označené **@ControllerAdvice** môžu obsahovať metódy s anotáciou **@ExceptionHandler**, ktoré budú aplikované na výnimky vyvolané v rámci viacerých kontrolerov.A screen shot of a computer program

Description automatically generated

3. ResponseStatusException

Od verzie Spring 5.0 je možné použiť triedu **ResponseStatusException** na programové vyvolanie výnimky s konkrétnym HTTP stavovým kódom. Tento prístup umožňuje priamo v kontroleri alebo v službe vyvolať výnimku, ktorá bude obsahovať správu a stavový kód HTTP.

A screen shot of a computer program

Description automatically generated

4. HandlerExceptionResolver

Spring poskytuje rozhranie **HandlerExceptionResolver**, ktoré umožňuje vlastné spracovanie výnimiek. Implementáciou tohto rozhrania môžete definovať vlastné správanie pre spracovanie výnimiek. **SimpleMappingExceptionResolver** a **DefaultErrorAttributes** sú príkladmi implementácií, ktoré Spring poskytuje.

Záver

Spring poskytuje flexibilný a mocný mechanizmus na spracovanie výnimiek, ktorý umožňuje vývojárom efektívne a elegantne riešiť chyby v aplikácii. Výber správneho prístupu závisí od konkrétnych potrieb aplikácie a preferencií vývojára.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Kedy má zmysel definovať EXCEPTION ako vlastnú?**

Definovanie vlastných výnimiek (custom exceptions) má zmysel v niekoľkých situáciách, ktoré prispievajú k lepšej čitateľnosti, údržbe a manipulácii s chybami v aplikácii. Tu sú kľúčové dôvody, prečo a kedy by ste mali zvážiť vytvorenie vlastných výnimiek:

1. Špecifické chybové stavy

Ak vaša aplikácia potrebuje zvládnuť špecifické chybové stavy, ktoré nie sú adekvátne reprezentované existujúcimi výnimkami v Jave alebo v rámci používaných knižníc. Vlastné výnimky umožňujú jasne definovať a rozlíšiť rôzne druhy chýb špecifických pre vašu aplikáciu.

2. Lepšia kontrola a manipulácia s chybami

Vlastné výnimky umožňujú definovať dodatočné informácie alebo kontext spojený s chybou, ako sú chybové kódy alebo detailnejšie popisy. Toto môže zlepšiť manipuláciu s chybami a umožniť konzistentnejšie spracovanie výnimiek naprieč aplikáciou.

3. Zlepšenie čitateľnosti a dokumentácie kódu

Použitie vlastných výnimiek môže zlepšiť čitateľnosť kódu a uľahčiť jeho dokumentáciu, keďže názvy a štruktúra vlastných výnimiek môžu odrážať konkrétne obchodné prípady alebo logiku aplikácie.

4. Zjednodušenie exception handlingu

Vlastné výnimky umožňujú centralizovať spracovanie špecifických druhov chýb. Napríklad, v Spring aplikácii môžete použiť **@ExceptionHandler** alebo **@ControllerAdvice** na zachytenie a spracovanie vlastných výnimiek špecifickým spôsobom, čím zjednodušujete kód potrebný na spracovanie chýb.

5. Podpora pre špecifické reakcie na chyby

Definovaním vlastných výnimiek môžete presne špecifikovať, aké akcie by mali byť vykonané pri určitých chybách, napríklad logovanie, posielanie upozornení alebo automatické opakovanie operácií. To umožňuje aplikácii reagovať flexibilnejšie na rôzne situácie.

Príklad:

A computer screen with text on it

Description automatically generated

Výše uvedený príklad **UserNotFoundException** ukazuje, ako môžete vytvoriť vlastnú výnimku, ktorá jasne identifikuje špecifický chybový stav (v tomto prípade nenájdenie používateľa) a umožňuje presnejšie spracovanie tejto situácie v aplikácii.

Vo všeobecnosti, ak výnimka, ktorú potrebujete spracovať, má špecifický kontext alebo by mala vyvolať špecifickú reakciu v aplikácii, má zmysel definovať pre ňu vlastnú triedu výnimky.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Pracujete aj s DOCKER alebo KUBERNETES?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako ste riešil komunikáciu medzi MICROSERVISMI (cross transakčnosť medzi viacerými mikroservismi)?

Riešenie komunikácie a transakčnosti medzi mikroservismi, najmä keď ide o operácie, ktoré vyžadujú konzistenciu a integritu dát naprieč viacerými službami, je jednou z kľúčových výziev v architektúre založenej na mikroservisoch. V tradičných monolitických systémoch sa transakcie spravujú pomocou lokálnych transakcií v rámci jednej databáze, čo umožňuje jednoduché zachovanie konzistencie dát. V rozdelenom prostredí mikroservisov, kde každá služba môže používať vlastnú databázu, je nutné nájsť iné spôsoby, ako zabezpečiť transakčnú integritu. Tu sú niektoré prístupy:

1. SAGA Pattern

SAGA je sekvenčná sada lokálnych transakcií, kde každá transakcia aktualizuje dáta v jednej službe a potom publikuje udalosť (event), ktorá spúšťa ďalšiu transakciu v inej službe. Ak jedna z transakcií zlyhá, SAGA spustí kompenzačné transakcie, ktoré vrátia systém do konzistentného stavu. Tento prístup môže byť implementovaný dvoma spôsobmi:

* **Orchestrácia:** Kde jedna centrálna služba (orchestrátor) riadi postupnosť transakcií a kompenzačných akcií.
* **Choreografia:** Kde služby komunikujú priamo pomocou udalostí bez centrálneho koordinátora, a každá služba vie, ako reagovať na prijaté udalosti.

2. 2PC (Two-Phase Commit)

Hoci 2PC je efektívny v monolitických databázach, jeho použitie v mikroservisovej architektúre je komplikované a môže viesť k problémom so škálovateľnosťou a dostupnosťou, pretože vyžaduje silnú synchronizáciu medzi službami. Preto nie je v mikroservisoch často odporúčaný.

3. Eventual Consistency

Prijatie eventual consistency znamená, že systém zaručuje konzistenciu dát len po určitom čase, nie okamžite. Tento prístup sa často používa spolu s asynchrónnou komunikáciou (napríklad pomocou správ alebo udalostí), kde sa systém postupne "uzdravuje" a dosahuje konzistentný stav.

4. Compensating Transactions

Ako súčasť SAGA vzoru, kompenzačné transakcie sú operácie, ktoré "vrátia späť" efekty predchádzajúcej operácie v prípade, že nasledujúca operácia v sekvencii zlyhá. Toto je kľúčové pre zabezpečenie integrity dát bez potreby striktných transakcií.

5. Outbox Pattern

Outbox pattern je technika, kde zmeny dát a udalosti, ktoré majú byť publikované, sú najprv uložené v "outbox" tabuľke v databáze zdrojovej služby. Tieto udalosti sú potom asynchrónne prenášané do cieľovej služby, čo znižuje riziko straty správ a umožňuje zachovanie poradia udalostí.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Používal ste aj JCA? Co je to JEE Connector Architecture (JCA) ?

JEE Connector Architecture (JCA) je štandardná architektúra v platforme Java EE (Enterprise Edition), ktorá umožňuje bezpečnú integráciu Java aplikácií s externými systémami, ako sú aplikácie pre plánovanie zdrojov podniku (ERP), staršie informačné systémy (legacy systems) a iné systémy a siete. Cieľom JCA je zjednodušiť integráciu týchto externých systémov s aplikáciami bežiacimi na Java EE serveroch.

JCA poskytuje štandardizovaný spôsob pre vytváranie zdrojových adaptérov, ktoré sprostredkúvajú komunikáciu medzi Java EE aplikáciami a externými systémami. Tieto adaptéry sú navrhnuté tak, aby boli opakovateľne použiteľné pre rôzne aplikácie. Architektúra definuje nasledujúce hlavné komponenty:

1. **Resource Adapter**: Modul, ktorý implementuje logiku potrebnú na interakciu s externým systémom. Adaptér zdroja môže napríklad zahŕňať konektivitu s databázovým serverom, správou frontov správ alebo akýmkoľvek iným systémom.
2. **Application Server Contract**: Definuje, ako aplikačný server (napr. GlassFish, WildFly) komunikuje s adaptérom zdroja, vrátane správy transakcií, bezpečnosti a správy pripojení.
3. **Common Client Interface (CCI)**: Opcionálny API, ktorý poskytuje štandardizovaný spôsob prístupu k funkciám adaptéra zdroja pre aplikácie. CCI umožňuje vývojárom pracovať s rôznymi zdrojovými adaptérmi konzistentným spôsobom.

JCA teda umožňuje vývojárom efektívne integrovať Java EE aplikácie s rôznymi externými systémami bez potreby hlbokých znalostí o týchto systémoch, pričom sa zachováva vysoká úroveň bezpečnosti, škálovateľnosti a správy transakcií. Použitím JCA môžu organizácie zjednodušiť a zefektívniť proces integrácie, čo vedie k rýchlejšiemu vývoju a nasadeniu aplikácií.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Popíšte ako u Vás fungovalo CI/CD buildovanie.

CI/CD je skratka pre Continuous Integration (CI) a Continuous Delivery alebo Continuous Deployment (CD). Tieto procesy automatizujú kroky vo vývoji softvéru, najmä testovanie a nasadzovanie, čo umožňuje tímom rýchlejšie a efektívnejšie vydávanie nových aktualizácií a funkcií. Tu je popis toho, ako funguje CI/CD buildovanie:

1. **Continuous Integration (CI):**
   * **Integrácia zmien kódu:** CI začína v momente, keď vývojár urobí zmeny v kóde a nahraje (push) ich do repozitára (často na Git). Tento proces zahŕňa pravidelné zlučovanie zmeneného kódu do hlavnej vetvy repozitára, čo pomáha predchádzať konfliktom.
   * **Automatické buildy:** Keď sú zmeny nahrané do repozitára, CI server automaticky spustí build proces, ktorý kompiluje kód, aby sa overilo, že nové zmeny neprerušia žiadnu časť aplikácie.
   * **Testovanie:** Po úspešnom skompilovaní kódu sa automaticky spustia jednotkové a integračné testy. Toto je kritické na zabezpečenie kvality kódu a zistenie chýb skôr, ako sa dostanú do produkcie.
   * **Záznam a spätná väzba:** Ak v procese CI dojde k chybe (či už pri kompilácii alebo testovaní), systém upozorní vývojársky tím, aby mohli problém rýchlo opraviť. Ak prejde všetko hladko, tím dostane potvrdenie o úspechu.
2. **Continuous Delivery (CD):**
   * **Automatické nasadzovanie do testovacieho prostredia:** Po úspešnej integrácii a testovaní môže byť kód automaticky nasadený do testovacieho prostredia. To umožňuje ďalšie testovanie, ako sú akceptačné testy alebo manuálne overenie funkčnosti.
   * **Príprava na vydanie:** V Continuous Delivery sa kód po úspešnom testovaní pripraví na vydanie, ale vyžaduje manuálne schválenie na nasadenie do produkcie.
3. **Continuous Deployment (CD):**
   * **Automatické nasadzovanie do produkcie:** Ak vaše procesy zahŕňajú Continuous Deployment, po úspešnom prechode cez všetky predchádzajúce fázy (vrátane CI a testovacieho nasadenia) sa kód automaticky nasadí do produkčného prostredia bez potreby manuálneho schválenia.
   * **Monitorovanie a optimalizácia:** Po nasadení do produkcie je dôležité monitorovať aplikáciu a získavať spätnú väzbu od koncových používateľov, čo umožňuje rýchle iterácie a vylepšenia.

CI/CD procesy umožňujú tímom udržiavať vysokú kvalitu softvéru, rýchlo reagovať na chyby a nové požiadavky a znižovať riziká spojené s manuálnym nasadzovaním. Vývojári sa tak môžu sústrediť viac na vývoj funkcií namiesto rutinných úloh.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Buildovali ste aj BRANCHE?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Čo z JDK by ste použil ako NEBLOKUJÚCE nástroje na prístup k MULTITHREATS? (akú CLASSu použiť?)

Pri práci s multithreadingom v Jave a zameraním sa na neblokujúci prístup, by ste mali používať nástroje a triedy, ktoré podporujú súbežnosť bez nutnosti blokovania. To znamená, že vlákna môžu pokračovať vo vykonávaní operácií aj keď čakajú na nejaké zdroje alebo dáta. Tu sú niektoré z neblokujúcich tried, ktoré JDK poskytuje pre prácu s multithreadingom:

1. **Atomic variables (z balíčka java.util.concurrent.atomic)**: Triedy ako **AtomicInteger**, **AtomicLong**, **AtomicReference** atď., poskytujú neblokujúci spôsob práce s premennými, kde každá operácia na úpravu hodnoty je atomická a vykonaná bez potreby synchronizácie.
2. **Concurrent Collections (z balíčka java.util.concurrent)**: Kolekcie ako **ConcurrentHashMap**, **ConcurrentLinkedQueue** atď., sú navrhnuté pre vysokovýkonné súbežné prístupy bez potreby blokovania pri čítaní alebo písaní.
3. **CompletableFuture (z balíčka java.util.concurrent)**: Táto trieda vám umožňuje písať neblokujúci asynchrónny kód, ktorý môže efektívne spracovať výsledky operácií, keď sú k dispozícii, bez nutnosti blokovania vlákna.
4. **Locks (z balíčka java.util.concurrent.locks)**: Hoci mnohé zámky môžu byť blokujúce, **Lock** rozhranie poskytuje metódu **tryLock()**, ktorá umožňuje vláknu pokúsiť sa získať zámok bez blokovania. Ak zámok nie je dostupný, vlákno môže pokračovať v iných operáciách alebo skúsiť zámok získať neskôr.
5. **Non-blocking I/O (z balíčka java.nio.channels)**: Pre prácu so sieťovými operáciami môžete použiť non-blocking I/O kanály ako **SocketChannel** a **ServerSocketChannel**, ktoré umožňujú čítanie a písanie dát bez blokovania vlákna.

V závislosti od konkrétneho použitia v multithreadingovom kontexte si môžete z týchto tried vybrať tie, ktoré najlepšie vyhovujú vašim potrebám pre neblokujúci prístup a efektívnu súbežnosť.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aké sú princípy REST API?

Princípy REST API (Representational State Transfer API) sú základné koncepty a usmernenia, ktoré definujú, ako by mali byť navrhované webové služby pre dosiahnutie lepšej škálovateľnosti, výkonnosti, a interoperability medzi rôznymi systémami a aplikáciami na internete. REST je architektonický štýl, ktorý bol popísaný Royom Fieldingom vo svojej doktorandskej práci v roku 2000. Tieto princípy zahŕňajú:

1. Jednotné rozhranie (Uniform Interface)

Tento princíp je základom REST architektúry a zahŕňa používanie štandardizovaných operácií HTTP (GET, POST, PUT, DELETE, atď.) pre interakcie s webovými zdrojmi. Každý zdroj by mal byť identifikovateľný pomocou URI (Uniform Resource Identifier). Jednotné rozhranie uľahčuje interakciu medzi klientom a serverom a zvyšuje rozšíriteľnosť systému.

2. Bezstavovosť (Stateless)

Komunikácia medzi klientom a serverom je bezstavová, čo znamená, že každá požiadavka od klienta na server musí obsahovať všetky informácie potrebné na jej pochopenie a spracovanie, vrátane autentifikačných a autorizačných informácií. Server neuchováva žiadny stav klienta medzi požiadavkami, čo zvyšuje spoľahlivosť a škálovateľnosť služby.

3. Ukladanie do cache (Cacheable)

Odpovede na požiadavky by mali byť, kde je to možné, cacheovateľné. To znižuje zaťaženie servera a zlepšuje celkovú výkonnosť a rýchlosť aplikácie tým, že znižuje potrebu opakovaných požiadaviek na server pre rovnaké informácie.

4. Klient-server architektúra

REST oddeluje klienta (napríklad webovú aplikáciu) od servera (kde sa nachádzajú zdroje a dáta), čo umožňuje každej časti nezávisle sa vyvíjať a škálovať. Táto separácia zjednodušuje komponenty na strane klienta a servera, čo vedie k modulárnejšej a flexibilnejšej architektúre aplikácie.

5. Vrstvený systém (Layered System)

Architektúra môže obsahovať viacero vrstiev, kde každá vrstva má špecifickú funkciu, ako sú zabezpečenie, load balancing, cachovanie atď. Klient nemusí byť schopný rozlíšiť, či komunikuje priamo so serverom, alebo s intermediárnou vrstvou. Toto zlepšuje bezpečnosť a umožňuje viacero vrstiev medziacií, ktoré môžu zvyšovať výkonnosť a škálovateľnosť.

6. Kód na požiadanie (optional)

Tento princíp umožňuje, že server môže dočasne rozšíriť alebo prispôsobiť funkcionalitu klienta poslaním vykonateľného kódu (napr. skripty JavaScript), ktorý klient potom môže vykonať. Táto vlastnosť nie je povinná a je zriedkavo využívaná.

Tieto princípy sú navrhnuté tak, aby podporovali vytváranie spoľahlivých, flexibilných a škálovateľných webových aplikácií a služieb.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aké zásadné pravidla platia pri RESTFULL SERVICEoch?

Pri návrhu RESTful služieb sa odporúča dodržiavať niekoľko zásadných pravidiel, aby boli služby dobre navrhnuté, udržateľné a ľahko použiteľné. Tieto pravidlá vychádzajú z obecných princípov REST architektúry a sú zamerané na zlepšenie komunikácie medzi klientom a serverom. Tu sú kľúčové pravidlá pre RESTful služby:

1. Používajte HTTP metódy explicitne

RESTful API by malo explicitne využívať štandardné HTTP metódy pre manipuláciu so zdrojmi:

* **GET** pre získanie dát bez ich zmeny.
* **POST** pre vytvorenie nového zdroja.
* **PUT** alebo **PATCH** pre aktualizáciu existujúceho zdroja. PUT pre kompletnú aktualizáciu a PATCH pre čiastočnú aktualizáciu.
* **DELETE** pre odstránenie zdroja.

2. Buďte štátovo nezávislí (Stateless)

Každá požiadavka od klienta na server musí obsahovať všetky informácie potrebné na jej pochopenie a spracovanie, bez toho, aby si server musel pamätať predchádzajúci stav klienta. Toto zabezpečuje vyššiu spoľahlivosť a škálovateľnosť služby.

3. Používajte jednoznačné URI pre zdroje

Každý zdroj by mal byť jednoznačne identifikovateľný pomocou URI (Uniform Resource Identifier). Struktúra URI by mala byť logická a predvídateľná, aby uľahčila prácu vývojárom pri práci s API.

4. Zabezpečte samopopisujúce správy

Odpovede servera by mali obsahovať dostatočné informácie (napríklad statusové kódy HTTP, hlavičky a telo odpovede), aby klient mohol pochopiť výsledok svojej požiadavky bez potreby ďalších dotazov.

5. Umožnite cacheovanie odpovedí

Odpovede by mali byť, kde je to možné, označené ako cacheovateľné, aby sa znížila potreba opakovaných požiadaviek na server a zlepšila celková výkonnosť systému.

6. Používajte HATEOAS (Hypermedia As The Engine Of Application State)

HATEOAS je princíp, ktorý umožňuje klientom dynamicky navigovať po API pomocou hypermediálnych odkazov poskytnutých spolu s odpoveďami. Tento prístup podporuje objavovanie funkcií služby bez potreby predchádzajúceho znalosti jej štruktúry.

7. Poskytujte adekvátne statusové kódy

Pri odpovediach na požiadavky používajte štandardné HTTP statusové kódy, aby ste poskytli jasnú spätnú väzbu o výsledku operácie (napr. 200 OK, 404 Not Found, 500 Internal Server Error).

Dodržiavaním týchto pravidiel môžete navrhnúť RESTful služby, ktoré sú ľahko použiteľné, dobre škálovateľné a udržateľné.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako by si riešil handlovanie interrupting injections?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Čo je efektívnejšie ako JSON? (používa to Twitter napríklad)

<https://msgpack.org/index.html>  
A screenshot of a computer

Description automatically generated

Twitter a mnohé iné technologické spoločnosti často hľadajú efektívnejšie alternatívy k JSON-u, aby zlepšili výkon a efektívnosť pri práci s veľkými objemami dát. Jednou z takýchto alternatív, ktorú Twitter používa, je formát nazývaný "MessagePack".

MessagePack je binárny formát serializácie, ktorý je účinný z hľadiska veľkosti a rýchlosti. V porovnaní s JSON, ktorý je textový formát, MessagePack dokáže ukladať rovnaké dáta v menšom objeme, čo znamená, že potrebuje menej miesta na disku a menej šírky pásma pri prenose cez sieť. Okrem toho, keďže MessagePack je binárny, jeho analýza (parsing) a generovanie môže byť rýchlejšie, pretože vyžaduje menej konverzií a manipulácie so stringami.

Použitie MessagePack prináša výhody najmä v prostrediach, kde je kritická efektívnosť prenosu dát a rýchlosť ich spracovania, ako je napríklad distribuované výpočtové systémy, mikroslužby, mobilné aplikácie s obmedzeným pripojením na internet a výkonné webové aplikácie.

Pre vývojárov, ktorí programujú v Jave a dbajú na optimalizáciu kódu z hľadiska výkonu a minimálnej spotreby pamäte a CPU, môže byť prechod z JSON na MessagePack alebo podobný efektívnejší formát serializácie cenným krokom. Pri implementácii je dôležité dodržiavať princípy OOP a SOLID, aby sa zachovala čitateľnosť a udržateľnosť kódu. Napríklad, pri integrácii MessagePack do Java aplikácie, je vhodné vytvoriť abstrakciu pre serializačný mechanizmus, čo umožňuje jednoduchú zmenu serializačného formátu v budúcnosti bez potreby zásadných zmien v aplikácii.

Co je to REST skratka?

Skratka REST znamená "Representational State Transfer", čo v preklade znamená "Prenos reprezentatívneho stavu". Tento termín bol poprvé zavedený Royom Fieldingom vo svojej doktorandskej práci v roku 2000 a opisuje architektonický štýl pre distribuované systémy, najmä webové služby. REST je navrhnutý tak, aby využíval existujúce protokoly a normy internetu, hlavne HTTP, a zjednodušil tak komunikáciu medzi klientmi a servermi. Princípy REST, ako bezstavovosť, jednotné rozhranie, a možnosť ukladania do cache, pomáhajú vytvárať výkonné, flexibilné a škálovateľné webové aplikácie a služby.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Hovorí Ti niečo TOAST v PostgreeSQL?

Áno, TOAST (The Oversized-Attribute Storage Technique) je mechanizmus v PostgreSQL, ktorý umožňuje efektívne ukladanie veľkých hodnôt atribútov, ktoré by inak prekročili maximálnu veľkosť bloku databázy, ktorá je štandardne nastavená na 8 KB. TOAST automaticky komprimuje a rozdelí veľké objekty na menšie časti, ktoré sú uložené oddelene od hlavnej tabuľky, a udržiava odkazy na tieto časti v hlavnej tabuľke. Tento proces je pre užívateľa úplne transparentný.

TOAST sa používa pre dátové typy, ktoré môžu obsahovať veľké množstvo dát, ako sú textové reťazce, binárne dáta (bytea), alebo iné typy, ktoré môžu prekročiť veľkosť bloku. Mechanizmus TOAST umožňuje PostgreSQL efektívne spracovávať a ukladať veľké dáta bez nutnosti manuálneho rozdeľovania dát alebo použitia externých úložísk.

Pri vkladaní alebo aktualizácii riadku, ak veľkosť dát prekročí určitý prah, PostgreSQL automaticky použije TOAST na kompresiu a rozdelenie dát na menšie fragmenty, ktoré sú uložené v špeciálnych TOAST tabuľkách. Pri čítaní dát PostgreSQL automaticky rekonštruuje originálne dáta z týchto fragmentov. Tento proces umožňuje optimalizovať výkon a spotrebu miesta na disku pri práci s veľkými objemami dát.

TOAST tak značne rozširuje schopnosti PostgreSQL pri práci s veľkými dátami, zatiaľ čo minimalizuje overhead spojený s ich ukladaním a spracovaním.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako by si v JAVA spravil počítadlo z viacerých THREATS? (akú CLASS-u by si použil?)  
  
V Jave, keď potrebujete bezpečne zdieľať a meniť hodnotu počítadla medzi viacerými vláknami (threads), je dôležité zabezpečiť, aby operácie na počítadle boli atómové, teda nedeliteľné. To znamená, že v danom čase môže hodnotu počítadla meniť iba jedno vlákno, čím sa zabráni problémom súvisiacim s konkurenciou a súbežným prístupom.

Jednou z bežných tried používaných na tento účel je **AtomicInteger** z balíka **java.util.concurrent.atomic**. Táto trieda poskytuje metódy na atómové operácie, ako sú inkrementácia (**incrementAndGet()**), dekrementácia (**decrementAndGet()**), a nastavenie novej hodnoty (**getAndSet(int newValue)**), bez potreby explicitného synchronizovania kódu.

Príklad použitia AtomicInteger na implementáciu počítadla:

A computer screen shot of a program

Description automatically generated

V tomto príklade **AtomicInteger** automaticky zabezpečuje, že každá operácia s počítadlom je atómová, čo znamená, že je úplne izolovaná od ostatných operácií, ktoré sa súčasne vykonávajú v iných vláknach. Toto je kľúčové pre vytváranie bezpečných a spoľahlivých súbežných aplikácií v Jave.

Použitie **AtomicInteger** je vhodné pre jednoduché atómové operácie. Pre zložitejšie operácie alebo operácie zahŕňajúce viacero premenných by ste mohli potrebovať použiť vyššie úrovne synchronizácie alebo iné mechanizmy z balíka **java.util.concurrent**.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Z čoho sa skladá JAVA MEMORY?  
  
Java pamäť, alebo Java Virtual Machine (JVM) pamäť, sa skladá z niekoľkých oblastí, ktoré sú alokované pri štarte JVM. Tieto oblasti sú navrhnuté tak, aby optimalizovali vykonávanie Java aplikácií, riadili alokáciu a dealokáciu pamäte a minimalizovali potrebu manuálneho spravovania pamäte. Tu sú hlavné časti pamäte JVM:

1. Heap (Kopa)

Heap je oblasť pamäte, kde sa ukladajú inštancie tried (objekty) a polia. Je to centrálna časť pamäte pre dynamickú alokáciu, kde sa vytvárajú a ukladajú všetky objekty a ich polia. Heap sa ďalej delí na:

* **Young Generation (Mladá generácia)**: Rýchlo sa plní a ukladajú sa v nej novovytvorené objekty. Delí sa na ďalšie časti: Eden Space a dve Survivor Spaces (S0 a S1).
* **Old Generation (Stará generácia)**: Obsahuje objekty, ktoré prežili viacero cyklov garbage collection v mladej generácii. Tieto objekty sú považované za stabilnejšie a menej pravdepodobne sa dealokujú.
* **Permanent Generation (PermGen) alebo Metaspace (od Java 8)**: Ukladá metadáta tried, ako sú definície tried a metód. Od Java 8 bola PermGen nahradená Metaspace, ktorá sa nachádza mimo heap a využíva natívnu pamäť JVM.

2. Stack (Zásobník)

Každé vlákno v Java aplikácii má svoj vlastný stack, ktorý ukladá lokálne premenné, volania metód a čiastkové výsledky. Stack pracuje podľa princípu LIFO (Last In, First Out) a je rýchly, pretože prístup a alokácia pamäte sú veľmi efektívne.

3. Program Counter (PC) Register

PC register obsahuje adresu aktuálne vykonávanej inštrukcie JVM. Každé vlákno má svoj vlastný PC register.

4. Native Method Stack

Táto oblasť je určená pre natívne metódy, ktoré nie sú napísané v jazyku Java, ale napríklad v C alebo C++. Podobne ako Java Stack, aj Native Method Stack je zvlášť pre každé vlákno.

5. Metaspace (od Java 8)

Nahrádza Permanent Generation a ukladá metadáta tried a iné informácie súvisiace s triedami. Metaspace využíva priamo pamäť operačného systému, čo umožňuje dynamické rozširovanie bez pevne nastavenej maximálnej veľkosti.

Správa pamäte v JVM zahŕňa automatický garbage collection, ktorý pomáha znížiť riziko memory leak (únik pamäte) tým, že automaticky dealokuje objekty, na ktoré už neexistujú žiadne odkazy.

Opísaná štruktúra a komponenty Java Virtual Machine (JVM) pamäte sú všeobecne aplikovateľné pre HotSpot JVM, ktorá je najčastejšie používanou implementáciou JVM od spoločnosti Oracle a je súčasťou oficiálneho Java Development Kit (JDK). HotSpot JVM je známa svojou vysokou výkonnosťou, optimalizáciou vykonávania kódu (napr. Just-In-Time kompiláciou) a efektívnym spravovaním pamäte, vrátane automatizovaného garbage collection mechanizmu.

HotSpot JVM je štandardná implementácia, ktorá sa používa pre väčšinu Java aplikácií a je dostupná pre rôzne operačné systémy, vrátane Windows, Linux a macOS. Podrobnosti o pamäťovej štruktúre, ako je rozdelenie heapu na Young Generation, Old Generation, a (v starších verziách) Permanent Generation alebo (v novších verziách) Metaspace, sú špecifické pre HotSpot JVM.

Je dôležité poznamenať, že existujú aj iné implementácie JVM, ako sú OpenJ9 od Eclipse Foundation a GraalVM od Oracle, ktoré môžu mať odlišné pamäťové modely a správu pamäte. Tieto alternatívne JVM môžu ponúkať rôzne optimalizácie a funkcionality, ktoré môžu byť v niektorých scenároch výhodnejšie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako funguje JAVA HEAP MEMORY? (z akých častí sa skladá?)  
  
Java Heap Memory je časť pamäte Java Virtual Machine (JVM), kde sa ukladajú runtime dáta aplikácií, najmä objekty a inštancie tried. Heap memory je zdieľaná medzi všetkými vláknami aplikácie a je hlavným zdrojom pre dynamickú alokáciu pamäte. Správa tejto pamäte zahŕňa alokáciu pamäte pre nové objekty a uvoľňovanie pamäte pre objekty, na ktoré už neexistujú odkazy, prostredníctvom procesu zvaného Garbage Collection (GC).

Java Heap Memory je typicky rozdelená do troch hlavných oblastí (generácií), ktoré pomáhajú optimalizovať správu pamäte a efektívnosť Garbage Collection:

1. Young Generation

* **Eden Space:** Väčšina novovytvorených objektov je alokovaná v Eden Space. Keď táto oblasť zaplní, vyvolá sa menší GC, známy ako Minor GC, ktorý čistí objekty, ktoré už nie sú dosiahnuteľné (žiadne odkazy na ne).
* **Survivor Spaces (S0 a S1):** Po každom Minor GC, preživšie objekty z Eden Space a jedného z Survivor Spaces sú presunuté do druhého Survivor Space. Objekty, ktoré prežijú dostatočný počet Minor GC cyklov, sú potom presunuté do Old Generation.

2. Old Generation (Tenured Generation)

* Obsahuje objekty, ktoré prežili viaceré cykly Minor GC. Keďže sa predpokladá, že tieto objekty budú existovať dlhšie, Garbage Collection (Major GC alebo Full GC) v tejto oblasti sa vyvoláva menej často, ale je zvyčajne náročnejší na výkon, pretože prehľadáva a čistí veľkú časť pamäte.

3. Permanent Generation (PermGen) / Metaspace (od Java 8)

* **PermGen:** Táto oblasť bola používaná do Java 7 na ukladanie metadát tried, ako sú informácie o triedach a metódach, ktoré sú používané JVM. Kvôli pevne nastavenej veľkosti a možným Memory Leaks bola v Java 8 nahradená Metaspace.
* **Metaspace:** Od Java 8 nahradzuje PermGen a ukladá metadáta tried v natívnej pamäti operačného systému, čo umožňuje dynamické zväčšovanie bez pevných limitov (okrem hardvérových obmedzení a konfigurácie).

Správa Java Heap Memory a efektívnosť Garbage Collection sú kritické pre výkon a stabilitu Java aplikácií. JVM poskytuje rôzne Garbage Collectory (napr. Serial, Parallel, CMS, G1, ZGC, Shenandoah), ktoré môžu byť konfigurované na optimalizáciu výkonu aplikácie na základe jej potrieb a charakteristík behového prostredia.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Používali ste nejaké TOOLS na statickú analýzu kódu (napr. SONAR)?

Áno, statická analýza kódu je dôležitá súčasť vývoja softvéru, ktorá pomáha identifikovať potenciálne chyby, zraniteľnosti, neefektívne vzory a dodržiavanie kódovacích štandardov ešte pred spustením alebo testovaním aplikácie. Existuje mnoho nástrojov určených pre rôzne programovacie jazyky vrátane Javy. Tu sú niektoré z najpopulárnejších nástrojov na statickú analýzu kódu:

1. SonarQube (predpokladám, že ste mali na mysli tento nástroj, keď ste spomínali "SOLAR")

* SonarQube je veľmi obľúbený open-source platforma pre kontinuálnu inspekciu kvality kódu. Poskytuje komplexné správy o chybách, kódových zápachoch, duplikátoch, pokrytí testami a mnoho ďalšieho. Je integrovateľný s CI/CD pipeline a podporuje mnoho programovacích jazykov vrátane Javy.

2. FindBugs / SpotBugs

* SpotBugs, nasledovník FindBugs, je nástroj na identifikáciu bežných chýb v Jave kóde. Analyzuje bytecode Javových aplikácií na vyhľadávanie vzorov, ktoré zvyčajne korelujú s chybami.

3. Checkstyle

* Checkstyle je nástroj určený na kontrolu, či Java kód spĺňa preddefinovaný súbor kódovacích štandardov a pravidiel. Je to užitočné pre udržiavanie konzistencie kódu a dodržiavanie najlepších praktík.

4. PMD

* PMD je nástroj na statickú analýzu kódu, ktorý vyhľadáva vo zdrojovom kóde Javy bežné programátorské chyby, nevyužité zdroje, nadbytočný kód, zložité výrazy a iné problémy.

5. Error Prone

* Error Prone je nástroj integrovaný priamo do javac (Java kompilátora), ktorý vyhľadáva bežné chyby programátora v kóde podobne ako SpotBugs.

6. JaCoCo

* JaCoCo je nástroj na meranie pokrytia kódu testami, čo nie je priamo statická analýza, ale je dôležité pre zabezpečenie kvality kódu. Pokrytie testami je často sledovaným metrikom v procese statickej analýzy.

Tieto nástroje môžu byť použité samostatne alebo v kombinácii na zabezpečenie vysokej kvality kódu a dodržiavanie najlepších praktík vývoja softvéru. Integrácia týchto nástrojov do vývojového procesu, napríklad do Continuous Integration (CI) pipeline, pomáha včas odhaľovať a opravovať problémy, zvyšuje efektivitu vývoja a znižuje náklady na opravu chýb v neskorších fázach vývojového cyklu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aké SCOPE-y má HIBERNATE?  
  
Hibernate, ako rámec pre mapovanie objektov na relačné databázy (ORM) v Jave, nemá "scope" v rovnakom zmysle, ako je používaný termín v kontexte Spring alebo webových aplikácií. Namiesto toho, v kontexte Hibernate, hovoríme skôr o životných cykloch a kontextoch, v ktorých sú entity a session spravované. Avšak, existujú určité koncepty, ktoré sa týkajú správy stavu a životného cyklu entít a session, ktoré môžeme považovať za ekvivalent "scope" v Hibernate:

1. Session Scope

Session v Hibernate je hlavným rozhraním na komunikáciu s databázou. Spravuje kontext perzistencie entít a ich životný cyklus. Session môže byť otvorená na krátke obdobie (na vykonanie niekoľkých operácií a potom zavretá) alebo môže byť otvorená po dlhšiu dobu, závisí to od potrieb aplikácie. "Session scope" sa teda týka obdobia, počas ktorého je session otvorená a dostupná pre operácie s databázou.

2. Transaction Scope

Transakcie v Hibernate definujú hranice pre súbor operácií s databázou, ktoré by mali byť vykonané spoločne. Transakcia môže byť začatá a ukončená v rámci jednej session, a všetky operácie vykonané medzi týmito bodmi budú buď všetky potvrdené alebo všetky vrátené späť ako jeden logický blok. "Transaction scope" určuje, ktoré operácie sú súčasťou danej transakcie.

3. Persistence Context Scope

Persistence context je sada všetkých entít, ktoré sú aktuálne spravované v rámci jednej session. Entita je v persistence context "managed" alebo "detached". "Managed" entity sú automaticky synchronizované s databázou pri commitovaní transakcie, kým "detached" entity už nie sú súčasťou persistence context a zmeny na nich nie sú automaticky propagované do databázy. Scope persistence context teda odkazuje na životný cyklus, počas ktorého je entita spravovaná a sledovaná Hibernate session.

4. Application Scope

Aj keď Hibernate priamo nepracuje s pojmom "application scope" ako takým, aplikácie často používajú entity a session v rámci širšieho kontextu aplikácie. Napríklad, môžete mať entitu, ktorá je používaná a spravovaná cez rôzne časti aplikácie, a jej stav a perzistencia sú udržiavané konzistentne naprieč viacerými session a transakciami.

Je dôležité správne pochopiť a navrhnúť, ako budú session, transakcie a persistence context použité v aplikácii, aby sa zabezpečila efektívnosť, konzistencia dát a výkon aplikácie. Rozhodnutia o scope a životnom cykle týchto komponentov majú priamy vplyv na správanie aplikácie a jej schopnosť správne pracovať s databázou.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aké typy CACH-e má HIBERNATE?  
  
Hibernate podporuje dva hlavné typy cache s cieľom zlepšiť výkon aplikácie tým, že minimalizuje počet dopytov do databáze. Tieto dva typy cache sú:

1. First Level Cache (Cache prvej úrovne)

* **Popis:** Cache prvej úrovne je automaticky povolená v Hibernate a je asociovaná s **Session**. Každá **Session** má svoju vlastnú cache prvej úrovne, ktorá ukladá entity, s ktorými bola pracovaná počas života tejto **Session**. Táto cache pomáha znižovať počet databázových dopytov pre entity, ktoré boli už načítané v rámci aktuálnej **Session**.
* **Použitie:** Cache prvej úrovne je najužitočnejšia pri opätovnom prístupe k entitám v rámci jednej **Session**. Nemá vplyv na dopyty medzi rôznymi **Session**.

2. Second Level Cache (Cache druhej úrovne)

* **Popis:** Cache druhej úrovne je voliteľná a konfigurovateľná na úrovni aplikácie alebo "SessionFactory". Ak je povolená, môže byť zdieľaná medzi rôznymi **Session** v rámci tej istej aplikácie. Cache druhej úrovne môže ukladať dáta na rôzne úrovne, vrátane entít, zbierok, dotazov a dokonca aj prirodzených identifikátorov.
* **Použitie:** Cache druhej úrovne je vhodná pre dáta, ktoré sa často nezmenia, sú relatívne statické alebo sú často pristupované cez rôzne **Session**. Pomáha značne znížiť počet dopytov do databáze na celoaplikačnej úrovni.

Query Cache (Cache dotazov)

* **Popis:** Cache dotazov je špecifický typ cache druhej úrovne, ktorý ukladá výsledky dotazov. Táto cache vyžaduje, aby bola povolená aj cache druhej úrovne, pretože výsledky dotazov odkazujú na entity uložené v cache druhej úrovne.
* **Použitie:** Je užitočná pre aplikácie, kde sa opakovane vykonávajú špecifické dotazy a očakávané výsledky sa často nezmenia.

Kľúčové Body:

* **Konfigurácia:** Cache druhej úrovne a cache dotazov vyžadujú explicitnú konfiguráciu a integráciu s externými cache poskytovateľmi ako Ehcache, Infinispan, alebo Hazelcast.
* **Manažment:** Efektívne využívanie cache v Hibernate znamená správny výber, ktoré dáta cachovať, a konfiguráciu cache politík (napr., invalidácia cache, čas života záznamov v cache, atď.), aby sa zabezpečilo, že cache zlepšuje výkon bez negatívneho vplyvu na konzistenciu dát.

Výber a konfigurácia cache stratégií v Hibernate sú kľúčové pre optimalizáciu výkonu aplikácie, a mali by byť pečlivo zvážené v kontexte špecifických potrieb a charakteristík aplikácie.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Na čo slúži OFFSET v KafkaMQ?

V Apache Kafka, ktorý je distribuovaným streamovacím platformom, **offset** predstavuje unikátny identifikátor pre každú správu (známu ako záznam) v partition (časti) topicu. Offset slúži na určenie pozície správy v partition. Vďaka tomu môžu spotrebitelia (consumers) presne vedieť, ktoré správy už boli prečítané a od ktorého miesta v partition majú začať čítať, aby získali nové správy.

Kafka uchováva správy v poradí, v akom boli pridané do partition, a každá správa v partition má priradený sekvenčný offset. Offsety sú specifické pre každú partition a sú číselné hodnoty, ktoré postupne rastú.

Použitie offsetu umožňuje KafkaMQ poskytnúť dva kľúčové vlastnosti:

1. **At-least-once delivery**: Spotrebitelia môžu potvrdiť spracovanie správ až po ich úspešnom spracovaní. Ak dojde k zlyhaniu, môžu znovu začať čítať od posledného potvrdeného offsetu, čím sa zabezpečí, že žiadna správa nebude stratená.
2. **Replayability**: Vďaka explicitným offsetom môžu spotrebitelia ľahko začať čítať správy od konkrétneho bodu v histórii, čo umožňuje napríklad opätovné spracovanie historických dát pre analýzu alebo debugovanie.

Offsety sú základným stavebným blokom pre podporu "distributed log" paradigmy v Kafke, kde každý spotrebiteľ môže nezávisle sledovať svoj postup cez log správ. Kafka poskytuje API na uchovávanie offsetov, buď automaticky prostredníctvom Kafky samotnej alebo externým systémom pre detailnejšiu kontrolu. Toto umožňuje aplikáciám navrhnutým s ohľadom na výkon a minimálne používanie pamäte a CPU efektívne spravovať postup pri spracovaní správ.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aký je rozdiel medzi virtualizáciou a DOCKERizáciou?

Virtualizácia a Dockerizácia (používanie Dockeru) sú dve technológie používané na izoláciu a spúšťanie aplikácií v izolovanom prostredí, ale fungujú na odlišných princípoch a majú rozdielne charakteristiky.

Virtualizácia

Virtualizácia umožňuje spúšťať viacero operačných systémov na jednom fyzickom hardvére paralelne, pričom každý operačný systém funguje ako nezávislý virtuálny stroj (VM). Táto technológia sa opiera o hypervízor, ktorý je softvérová (alebo hardvérová) vrstva zodpovedná za distribúciu hardvérových zdrojov medzi rôzne VM. Každý virtuálny stroj má svoj vlastný operačný systém, aplikácie a nezávislé knižnice, čo znamená, že z hľadiska zdrojov sú relatívne náročné.

***Výhody:***

* Silná izolácia medzi virtuálnymi strojmi.
* Možnosť spúšťať rôzne operačné systémy na jednom fyzickom serveri.

***Nevýhody:***

* Vyššia spotreba zdrojov, pretože každý VM spúšťa plný operačný systém.
* Väčšia režia z hľadiska úložného priestoru a pamäte.

Dockerizácia (používanie Dockeru)

Docker využíva kontajnerizáciu, technológiu, ktorá umožňuje balenie aplikácií a ich závislostí do štandardizovaných jednotiek pre softvérový vývoj známych ako kontajnery. Na rozdiel od virtuálnych strojov, ktoré virtualizujú celý operačný systém, kontajnery zdieľajú rovnaký operačný systém hostiteľa a izolujú iba aplikácie a ich prostredie. To vedie k menšej režii a umožňuje rýchlejšie spustenie a väčšiu efektivitu využívania zdrojov.

***Výhody:***

* Menšie využitie zdrojov v porovnaní s VM, pretože kontajnery zdieľajú jadro operačného systému hostiteľa.
* Rýchlejšie spustenie, pretože nie je potrebné spúšťať celý operačný systém.
* Jednoduchšie spravovanie závislostí a konzistentné prostredie naprieč vývojovým a produkčným prostredím.

***Nevýhody:***

* Menšia izolácia v porovnaní s virtuálnymi strojmi, pretože všetky kontajnery zdieľajú jadro hostiteľského systému.
* Obmedzené na aplikácie, ktoré môžu bežať na rovnakom operačnom systéme, keďže kontajnery zdieľajú jadro operačného systému hostiteľa.

Zhrnutie

Hlavný rozdiel medzi virtualizáciou a Dockerizáciou (kontajnerizáciou) spočíva v úrovni izolácie a efektívnosti. Virtualizácia poskytuje silnú izoláciu na úkor väčšieho využitia zdrojov, zatiaľ čo Dockerizácia ponúka efektívnejšie využitie zdrojov a rýchlejšie spustenie s menšou, ale stále dostatočnou úrovňou izolácie pre mnohé využitia. Výber medzi týmito dvoma prístupmi závisí od špecifických potrieb projektu, ako sú požiadavky na bezpečnosť, výkon a infraštruktúra.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ako funguje REACTIVE JAVA?

Reactive Java, často spojená s Reactor knižnicou (ako je Project Reactor, ktorý je súčasťou Spring WebFlux) alebo RxJava, je programovací model založený na reaktívnom programovaní, ktorý umožňuje efektívne spracovávanie asynchrónnych dátových tokov. Tento model je zameraný na spracovanie udalostí, dátových tokov a implementáciu non-blocking back pressure. Hlavné princípy a koncepty reaktívneho programovania v Jave zahŕňajú:

1. Asynchrónne Dátové Toky

* V reaktívnom programovaní sa dáta považujú za toky udalostí, ktoré môžu byť generované a spracované asynchrónne. Toto umožňuje aplikáciám efektívne reagovať na dáta v reálnom čase, ako prichádzajú.

2. Non-blocking Back Pressure

* "Back pressure" je mechanizmus, ktorý umožňuje konzumentom kontrolovať rýchlosť, akou výrobcovia generujú dáta. Tým sa zabráni preťaženiu konzumentov a potenciálnej strate dát. V reaktívnom programovaní je back pressure dynamický a non-blocking, čo znamená, že konzumenti môžu asynchrónne signalizovať výrobcom, akú rýchlosť produkcie dát sú schopní spracovať.

3. Reaktívne Streamy

* Reaktívne programovanie v Jave využíva API reaktívnych streamov, ktoré definuje štyri základné interfacy: **Publisher**, **Subscriber**, **Subscription** a **Processor**. Tieto interfacy umožňujú vytváranie komponovateľných asynchrónnych dátových tokov.

4. Reaktívne Operátory

* Knihovnice ako RxJava alebo Project Reactor poskytujú bohatú sadu operátorov na transformáciu, filtrovanie, zlučovanie a iné manipulácie s dátovými tokmi. Tieto operátory umožňujú vyjadriť zložité asynchrónne logiky prúdenia dát deklaratívnym spôsobom.

5. Integrácia s Existujúcimi Frameworkmi

* Reaktívne programovanie je čoraz viac integrované do populárnych Java frameworkov, ako je Spring Framework cez Spring WebFlux, ktorý umožňuje vytváranie plne reaktívnych webových aplikácií.

Príklad použitia:

Predstavte si asynchrónne spracovanie HTTP požiadaviek v webovej aplikácii, kde každá požiadavka môže byť spracovaná nezávisle a asynchrónne bez blokovania serverových zdrojov. Reaktívne programovanie umožňuje efektívne spracovávanie týchto požiadaviek s dynamickým back pressure, zlepšujúc tak výkon a škálovateľnosť aplikácie.

Reaktívne programovanie predstavuje paradigmu zameranú na asynchrónne spracovanie dátových tokov s plnou podporou back pressure, čo je ideálne pre vývoj vysoko reaktívnych a škálovateľných aplikácií v prostredí súčasnej webovej a cloudovej architektúry.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Nejaký zaujímavý feature od JAVA 9+?  
  
Od uvedenia Java 9 sa do Javy pridalo niekoľko zaujímavých funkcií, ktoré zlepšili výkon, bezpečnosť a čitateľnosť kódu. Tu je niekoľko kľúčových funkcií a vylepšení, ktoré boli zavedené v Jave 9 a neskorších verziách:

Java 9

* **Modulový systém (Project Jigsaw):** Možno najvýznamnejšia zmena v Jave 9, ktorá priniesla modulový systém, umožňujúci lepšiu štruktúru aplikácií prostredníctvom modulov s jasnými závislosťami.
* **JShell:** Interaktívny REPL nástroj (Read-Eval-Print Loop) na experimentovanie s javovským kódom v reálnom čase.
* **Private Interface Methods:** Umožňuje definovať privátne metódy v rozhraniach, čím sa zlepšuje opätovná použiteľnosť kódu.

Java 10

* **Local-Variable Type Inference:** Zaviedlo kľúčové slovo **var**, ktoré umožňuje beztypovú deklaráciu lokálnych premenných a zjednodušuje kód.
* **Application Class-Data Sharing (CDS):** Zlepšuje štartovací čas aplikácie a znižuje spotrebu pamäte JVM.

Java 11

* **HttpClient API (štandardné):** Nahradzuje zastaranú **HttpURLConnection** a poskytuje moderné, výkonné a flexibilné riešenie pre HTTP volania.
* **Launch Single-File Source-Code Programs:** Umožňuje priamy beh súborov zdrojového kódu bez kompilácie.

Java 12

* **Switch Expressions (Preview):** Zjednodušuje **switch** výraz tým, že umožňuje viacnásobné výrazy a vrátenie hodnoty priamo zo **switch** príkazu.
* **Shenandoah: A Low-Pause-Time Garbage Collector:** Nový typ zberača odpadkov zameraný na minimalizáciu pauz.

Java 13

* **Text Blocks (Preview):** Uľahčuje prácu so stringami na viacerých riadkoch, čím zjednodušuje vytváranie JSON, HTML a SQL dotazov priamo v Jave.

Java 14 a neskoršie

* **Records (Preview in 14, standard in 16):** Umožňuje jednoduchšiu implementáciu dátových nosičov bez potreby boilerplate kódu.
* **Pattern Matching for instanceof (Preview in 14, standard in 16):** Zlepšuje čitateľnosť a zjednodušuje kód pri kontrole typov a ich následnom pretypovaní.
* **Sealed Classes (Preview in 15, standard in 17):** Umožňuje obmedziť, ktoré iné triedy môžu byť potomkami danej triedy.

Tieto funkcie a vylepšenia značne zlepšujú efektivitu vývoja, výkon a možnosti Javy ako programovacieho jazyka.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Píšete aj dokumentáciu k SW?

**Treba pisat aj dokumentaciu ku kodu?**

Áno, písanie dokumentácie kódu je dôležitým aspektom softvérového vývoja, ktorý prispieva k lepšej čitateľnosti, údržbe a rozširiteľnosti softvéru. Dokumentácia pomáha iným vývojárom (a niekedy aj vám v budúcnosti) rýchlejšie porozumieť účelu a fungovaniu kódu, čo uľahčuje jeho údržbu a rozvoj. Tu sú hlavné dôvody, prečo by ste mali písať dokumentáciu ku kódu:

1. Zlepšenie čitateľnosti a pochopiteľnosti

* Komentáre a dokumentácia v kóde poskytujú prehľad o tom, čo konkrétne časti kódu robia, prečo boli niektoré rozhodnutia prijaté a ako sú rôzne časti systému prepojené. To uľahčuje novým a existujúcim členom tímu rýchlejšie sa zorientovať.

2. Uľahčenie údržby a rozvoja

* Dobrá dokumentácia umožňuje ľahšie a rýchlejšie identifikovať potrebné zmeny alebo opravy v kóde. To je obzvlášť užitočné v situáciách, keď pôvodný autor kódu už nie je dostupný alebo keď pracujete na veľkých a zložitých projektoch.

3. Podpora najlepších praktík a štandardov

* Písanie dokumentácie podporuje dodržiavanie štandardov čistého kódu a dobrých programovacích praktík. To zahŕňa aj vysvetlenie zložitých algoritmov, ktoré môžu byť inak ťažko pochopiteľné.

4. Zjednodušenie revízie kódu

* Dokumentácia umožňuje ľahšie a efektívnejšie revízie kódu, pretože recenzenti môžu rýchlejšie pochopiť účel a očakávané správanie kódu, čo vedie k presnejším a konštruktívnejším spätným väzbám.

5. Zlepšenie testovania a ladenia

* Komentáre v kóde môžu poskytnúť užitočné informácie pre vývojárov, ktorí testujú alebo ladia systém, a pomôcť im lepšie porozumieť, ako by mali jednotlivé časti systému fungovať.

Ako písať efektívnu dokumentáciu

* **Komentujte zložité časti kódu**: Poskytnite vysvetlenie zložitých algoritmov alebo rozhodnutí, ktoré nie sú na prvý pohľad jasné.
* **Používajte konzistentný štýl**: Dodržiavajte konzistentný štýl písania dokumentácie a komentárov, aby bola dokumentácia ľahko čitateľná a pochopiteľná.
* **Aktualizujte dokumentáciu**: Udržujte dokumentáciu aktuálnu s najnovšími zmenami v kóde. Zastaraná dokumentácia môže byť viac škodlivá ako žiadna dokumentácia.
* **Používajte nástroje na generovanie dokumentácie**: Existujú nástroje, ako je Javadoc pre Java, ktoré môžu automaticky generovať dokumentáciu z komentárov v kóde, čo zjednod

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Kedy má podľa teba zmysel písať unit testy?

Písanie jednotkových testov (unit testov) má zmysel v mnohých situáciách počas vývojového cyklu softvéru. Tieto testy sú zamerané na overenie funkcionality najmenších častí kódu, ako sú funkcie alebo metódy, nezávisle od zvyšku systému. Tu sú kľúčové momenty, kedy je písanie jednotkových testov obzvlášť užitočné:

1. Pri Implementácii Novej Funkcionality

* **Testom Riadený Vývoj (TDD)**: Pri TDD najprv napíšete jednotkové testy, ktoré definujú, ako by mala nová funkcionalita fungovať, a až potom napíšete kód, ktorý tieto testy splní. Tento prístup zabezpečuje, že váš kód spĺňa všetky požiadavky ešte predtým, než sa pustíte do jeho detailného implementovania.

2. Pri Refaktorizácii

* Jednotkové testy umožňujú refaktorizovať existujúci kód s dôverou, že nezavediete nové chyby. Ak vaše zmeny spôsobia zlyhanie existujúcich testov, viete, že ste niečo pokazili.

3. Pri Oprave Chýb

* Keď nájdete chybu, je dobré napísať jednotkový test, ktorý túto chybu reprodukuje. Potom môžete chybu opraviť a použiť test na overenie, že chyba bola naozaj odstránená a že sa v budúcnosti neobjaví znova.

4. Na Zabezpečenie Kvality Kódu

* Jednotkové testy pomáhajú zabezpečiť, že váš kód robí to, čo má, a že každá časť kódu pracuje správne nezávisle od ostatných. To je dôležité pre dlhodobú udržateľnosť a kvalitu softvéru.

5. Pre Dokumentáciu Kódu

* Jednotkové testy môžu slúžiť ako forma živej dokumentácie. Keďže ukazujú, ako sa majú rôzne časti kódu používať, môžu byť užitočné pre nových členov tímu alebo pre vývojárov, ktorí sa k projektu vracajú po dlhšej dobe.

6. Pri Integrácii Nových Členov Do Tímu

* Jednotkové testy umožňujú novým vývojárom, aby sa rýchlejšie zorientovali v kóde, pretože poskytujú konkrétne príklady toho, ako by mali rôzne časti systému fungovať.

Zhrnutie

Jednotkové testy sú neoceniteľným nástrojom pre zabezpečenie kvality, udržateľnosti a rozširiteľnosti softvéru. Majú zmysel v takmer každej fáze vývojového cyklu, najmä ak sa zameriavate na vysokú kvalitu a spoľahlivosť vášho produktu. Ich implementácia by mala byť základnou súčasťou vašej vývojovej stratégie, v súlade s princípmi OOP a SOLID, aby sa maximalizovala modularita, flexibilita a testovateľnosť vášho kódu.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Radšej by si debugoval, alebo písal test?  
  
Vo vývoji softvéru, rozhodnutie medzi debugovaním a písaním testov závisí od konkrétneho kontextu a cieľov. Obe aktivity sú kľúčové aspekty vývojového cyklu, ale slúžia na riešenie rôznych problémov. Tu je niekoľko úvah, ktoré by vám mohli pomôcť rozhodnúť, čo je v danej situácii lepšie.

Debugovanie

* **Okamžitá Oprava**: Debugovanie je proces hľadania a odstraňovania chýb alebo "bugov" vo vašom kóde. Je nevyhnutné, keď váš program nepracuje správne a potrebujete rýchlo nájsť a opraviť konkrétne problémy.
* **Porozumenie Kódu**: Debugovanie môže pomôcť lepšie pochopiť, ako váš program funguje, najmä ak pracujete na časti kódu, ktorá bola napísaná niekým iným alebo keď sa vraciate k svojmu starému kódu.

Písanie Testov

* **Prevencia Chýb**: Testy (jednotkové testy, integračné testy atď.) sú navrhnuté tak, aby automaticky overovali správanie vášho kódu a zabezpečovali, že všetko funguje podľa očakávaní. Písanie testov môže predísť chybám skôr, než sa dostanú do produkčného prostredia.
* **Refaktorizácia a Údržba**: Testy poskytujú bezpečnostnú sieť, ktorá umožňuje refaktorizáciu a rozširovanie kódu s dôverou, že zmeny nezavedú nové chyby.

Kedy je Lepšie Čo?

* **Na začiatku vývoja** alebo pri implementácii nových funkcií môže byť výhodnejšie napísať testy, ktoré definujú očakávané správanie. Toto je známe ako testom riadený vývoj (TDD). TDD vám pomáha udržať kód čistý a zabezpečuje, že všetko funguje ako má od začiatku.
* **Pri existujúcom kóde s chybami** je často nevyhnutné použiť debugovanie na lokalizáciu a opravu špecifických problémov. Po oprave chýb by ste mali napísať testy na overenie opravy a zabezpečenie, že tá istá chyba sa neobjaví znova.

Najlepší Prístup

Ideálny prístup kombinuje oboje: používanie debugovania na identifikáciu a riešenie okamžitých problémov a písanie testov na zabezpečenie dôveryhodnosti a udržateľnosti kódu do budúcnosti. Tento prístup zabezpečí, že váš softvér bude nielen fungovať správne v krátkodobom horizonte, ale tiež bude ľahšie udržateľný a rozširiteľný v dlhodobom meradle.

V kontexte SOLID a OOP, písanie testov podporuje princípy ako **Single Responsibility Principle** (SRP) a **Open/Closed Principle** (OCP), keďže dobre navrhnuté testy a kód, ktorý je ľahko testovateľný, často vyžadujú, aby triedy mali jednu zodpovednosť a boli otvorené pre rozšírenie, ale zatvorené pre zmeny.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Aká je pridaná hodnota observables (výhody oproti promises)?

Observables a Promises sú dva rôzne prístupy k asynchrónnemu programovaniu v jazykoch ako JavaScript alebo TypeScript, ktoré sa dajú použiť aj v kontexte Java prostredníctvom príslušných knižníc alebo paralel. Hoci sa obe technológie používajú na zvládanie asynchrónnych operácií, majú zásadné rozdiely, ktoré im dávajú rôzne výhody.

Výhody Observables oproti Promises:

1. **Viaceré Hodnoty časom**: Promise sa vyrieši alebo zamietne s jednou hodnotou a je to jednorázová operácia. Naopak, Observable môže emitovať viacero hodnôt (vrátane nula) cez čas, čo je ideálne pre streamovanie dát alebo akékoľvek operácie, kde sa očakáva viacero hodnôt v časovej postupnosti.
2. **Zrušiteľnosť**: Observable operácie sú zrušiteľné, čo znamená, že ak sa asynchróna operácia stane nepotrebnou pred jej dokončením, môže byť zastavená. To je užitočné na uvoľnenie zdrojov a zamedzenie nepotrebnému spracovaniu. Promises v základnej forme túto funkcionalitu nepodporujú.
3. **Operátory**: Observables podporujú širokú škálu operátorov (ako map, filter, concat, flatMap a ďalšie), ktoré umožňujú komplexné spracovanie a kombinovanie dátových tokov predtým, ako sú poskytnuté konečnému odberateľovi. Táto flexibilita je obzvlášť užitočná pri práci s komplexnými asynchrónnymi tokmi dát.
4. **Lepšie ovládanie asynchrónnosti**: Vďaka možnosti spracovať viacero hodnôt a používať operátory na ich transformáciu, Observables poskytujú lepšie ovládanie nad asynchrónnymi operáciami. To umožňuje lepšie riadenie toku dát a asynchrónnych udalostí v aplikácii.
5. **Lazy vyhodnotenie**: Observables sú "lazy" teda ich vyhodnotenie (spustenie asynchrónnej operácie) začína až v momente, keď na ne niekto odberá dáta (subscribe). To umožňuje presnejšie ovládanie nad tým, kedy asynchrónna operácia začne, a zároveň šetrí zdroje, ak operácia nie je potrebná.

Záver:

Observables ponúkajú viac flexibility a kontrolu pre asynchrónne programovanie, najmä v prípadoch, kde sú potrebné viaceré hodnoty, zrušiteľnosť, komplexné spracovanie dát alebo lazy vyhodnotenie. Ich použitie je vhodné v situáciách, kde sú tieto vlastnosti požadované alebo kde sa očakáva dynamický a streamovaný tok dát. Napriek tomu, Promises sú stále veľmi užitočné pre jednoduchšie asynchrónne úlohy s jednorazovým výsledkom. Výber medzi nimi závisí od konkrétnych potrieb aplikácie a preferencií vývojára.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Čo je to multicasting?  
  
  
Multicasting v kontexte programovania, a špeciálne pri práci s Observables, je proces, pri ktorom sa dáta z jedného zdroja distribuujú súčasne viacerým odberateľom (subscribers). To znamená, že všetci odberatelia dostávajú rovnaké dáta z jedného spoločného zdroja, namiesto toho, aby každý odberateľ vytváral vlastnú, nezávislú kópiu dátového toku. Tento prístup je užitočný v situáciách, kde chcete efektívne zdieľať výsledky drahých alebo zdrojovo náročných operácií medzi viacerými spotrebiteľmi dát.

Ako Multicasting Funguje s Observables

V základnom stave je Observable unicast, čo znamená, že každé volanie **.subscribe()** na Observable vytvorí novú inštanciu toho Observable, s vlastným nezávislým vykonávaním. Pri multicastingu, naopak, Observable distribuuje výsledky svojho vykonávania viacerým odberateľom, čím zdieľa svoje zdroje a výsledky.

Výhody Multicastingu

1. **Efektívnosť Zdrojov**: Multicasting umožňuje zníženie záťaže na zdroje tým, že sa vykonáva len jedna inštancia operácie, ktorá je potom zdieľaná medzi všetkými odberateľmi.
2. **Synchronizácia Dát**: Všetci odberatelia dostávajú dáta súčasne, čo uľahčuje synchronizáciu stavu medzi rôznymi časťami aplikácie.
3. **Zníženie Latencie**: Keďže dáta sú spracované a distribuované z jedného zdroja, môže to zlepšiť časy odozvy v aplikáciách, kde je to kritické.

Ako dosiahnuť Multicasting v Observables

V knižniciach ako RxJS, ktorá rozširuje koncept Observables, môžete dosiahnuť multicasting použitím operátorov ako **share()**, ktorý transformuje unicast Observable na multicast Observable využívajúc subjekty (subjects). Subject v RxJS je špeciálny typ Observable, ktorý môže multicastovať: môže odosielať hodnoty viacerým odberateľom súčasne.

Príklad

Predstavte si situáciu, kde máte dátový stream s finančnými informáciami, ktorý chcete zdieľať s viacerými komponentmi vo vašej aplikácii bez toho, aby ste pre každého vytvárali nové pripojenie k zdroju dát. Použitím multicastingu môžete tieto dáta efektívne distribuovať všetkým relevantným komponentom.

V OOP a SOLID kontexte, multicasting podporuje princípy ako Single Responsibility Principle (SRP) tým, že oddelí logiku distribúcie dát od ich generovania, a Interface Segregation Principle (ISP) tým, že poskytuje rozhrania prispôsobené potrebám rôznych odberateľov.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Máš skúsenosti s dizajn paternami?

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Vieš nám aspoň jeden opísať? (okrem Factory a Singleton)

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Máš skúsenosti s toolkitmi?Bottom of Form