Nagyvállalati szoftverfejlesztés Java alapon vizsga kérdések

Utoljára módosítva: 2020. április 17.

Készítette: Ráncsik Áron

Tartalomjegyzék

1. SOLID elvek nagyvállalati gyakorlatban

1.1. Mit jelentenek a következő fogalmak?

Encapsulation Egységbezárás, az adatok és a rajtuk végzett műveleteket, OOP esetén osztályokba zárjuk.

Abstraction Elhagyunk a kontextus számára nem fontos dolgokat, és csak az adott feladat számára lényeges információ kivonattal dolgozunk.

Amit nyerünk:

- Komplex dolgok egyszerű kezelése
- Modellezés
- Duplikáció elkerülése

Inheritance Öröklődés. Legerősebb kapcsolat két osztály között.

Előnyei:

- Kód újrafelhasználás.
- Modellezésben: általánosabb dolgokat ki lehet szervezni ősosztályba
- Ősosztály több, független kiterjesztésére alkalmas.

Polimporphism Többalakúság. Különböző típusok egy interfaceként való kezelése.

Megvalósítása:

- Függvény túlterhelés.
- Generikus programozás.
- (OOP) függvény felülírás leszármaztatottban.

1.2. Milyen relációkat ismersz? Mi az a kompozíció?

Inheritance Öröklődés. lásd: 1.1

Interface Implementation Interface implementáció. Hasonló az öröklődéshez, de gyengébb a kapcsolat. Interface nem tartalmaz implementációt.

Composition Mező, attribútum egy osztályban. Egy osztály egy másik osztályra hivatkozik. Létrehozás és megszüntetés is a létrehozó osztály feladata.

Aggregation Mező, attribútum egy osztályban. Egy osztály egy másik osztályra hivatkozik. Az osztály csak használni tudja ha kap egy példányt. Nem tartozik a felelősségébe a példány létrehozása.

Associacion Mező, attribútum egy osztályban. Egy osztály egy másik osztályra hivatkozik. Az osztály **opcionálisan** csak használni tudja a másik objektumot, de ha null lenne a hivatkozás akkor is működőképes marad nélküle.

Dependecy Egy osztály módosítása esetlegesen kihathat egy másik osztály módosítására. Egy osztály dependál, ráépít, támaszkodik egy másik osztályra.

1.3. Ismertesd a SOLID elveket

Single Responsibility Egy osztálynak egy felelőssége legyen. És ez a felelősség lehetőleg ebben az egy osztályban legyen egysége zárva.
Egy osztálynak egy oka legyen a változásra.
Előny:

- Könnyebben lehet kezelni az osztályokat változó környezetben. Robusztus.
- Kisebb lesz a kohézió egy osztályon belül. A felelősségek szétválaszthatóak lesznek.

Open/Closed Osztályok, entitások lehetőleg legyenek nyitottak a bővítésre és zártak a módosításokra.
Zártság célja, hogy az osztályok módosítását elkerüljük. Arra törekszünk ne keljen az osztályt módosítani.
Nyíltság: Arra törekszünk, hogy bővítés esetén minimális módosításra legyen szükség.
Megoldás: Használjunk absztrakciót.

Liskov substitution Liskov féle helyettesítési szabály. Leszármaztatott osztály csak olyan módosításokat csinálhat ami nem változtatja az ősosztály felelősségét és működését. A leszármazott osztály példány helyettesíthető legyen az ősosztály példánnyal. Ha ősosztályként tekintek egy leszármaztatott osztályra nem szabad, hogy különbséget lássak, továbbra is működjön úgy ahogy egy ősosztály példány is működne. Feltételek:

- **Metódus szignatúrák egyezése** (Java nyelvben alap).
- **Preconditions.** Paraméterként átadott adatokkal kapcsolatos elvárások megtartása, csak erősíteni lehet. Pl. ha ősosztály működik null paraméterrel a akkor a leszármaztatott is működjön.
- **Postconditions.** A visszaadott eredmény nem szegheti meg az ősosztály által nyújtott garanciákat.

– **Invariánsok.** Ilyen értéket nem módosíthatjuk meg. Pl. Ha az ősosztály egy értéket nem módosított egy bizonyos metódus hívásra akkor lehetőleg a leszármazott se módosítsa azt.
– **History constarint.** A leszármazott nem módosíthatja olyan formában a belső állapotot ami megzavarja az ősosztály működését.

Interface segregation Klienseket nem kéne olyan függőségekre kényszeríteni amire nincs szükségük. Pl. Interface rengeteg metódussal, de ebből csak néhányat szeretnénk használni. Szerep alapú interfacek legyenek.

C++ esetén: header interface helyett szerep interface. Lustaságból lehetne a header fájlt interfacnek használni, de ne tegyük ezt.

Előny:
– Kevésbé lesz egybefüggő a kód.
– A változásokra kevésbé lesz érzékeny az implementáló kód.

Dependency inversion Magas-szintű moduloknak nem lenne szabad függeniük az alacsony-szintű moduloktól. Mind kettőnek absztrakcióktól kellene függeniük.

Régi szokás (volt) magas-szintű modulokat alacsony-szintű modulokra építeni. Ehelyett sokkal jobb, ha egy absztrakciós réteget hozunk létre a magas- és az alacsony-szintű réteg közé, tehát interfaceektől függjenek. Egy magas-szintű modul alacsony-szintű függőségeit pedig konstruktoron keresztül oldjuk fel.

Ebben az esetben viszont az a kérdés merül fel, hogy a függőségeket melyik rétegben példányosítsunk, erre megoldás az Inversion Of Control (röviden IoC) container. bővebben: 6.1

1.4. Mit jelentenek a következők?

Demeter szabály Egy objektum "A„ kérhet szolgáltatásokat (metódus hívással) "B„ objektumtól, de "A„ objektumnak nem kéne "B„ objektumot csak kihasználva "B„ től elkérnie egy másik "C„ objektumot, hogy aztán az "A'a "B„-n keresztül, "'C„-től kérjen szolgáltatásokat. Ebben az esetbem "A„ ismeri a "B„ belső szerkezetét és ez nem jó.

Előny:
– Rejtett függőség elkerülése.
– Kezelhetőbbé adaptálhatóbbá válik az osztály.

Tell, don't ask principle Ha egy objektum több hívás végez egy másik objektumon egymás után, egy feladat elvégzésére akkor azt a másik objektumot "kérdezgeti",

ehelyett, a másik objektum egy metódushíváson keresztül kiszolgálhatnánk a hívó kódot.

Mondjuk meg egy objektumnak, hogy mit csináljon, ne pedig az adatait kérdezgessük le, hogy utána megcsináljuk.

KISS Keep it simple stupid. Amennyire lehet egyszerűsítsünk, ne bonyolítsuk feleslegesen a kódot.

YAGNI You ain't gonna need it. Ne fejlessz olyan dolgok amire nem lesz szükség.

Immutable és Value objektum Immutable:(C# esetén megfeleltethető a primitív, egszerű típus) nem megváltoztatható objektum. "final„ mező. Szálbiztos, egyszerűen érthető, de módosításhoz le kell másolni
Value: (C# esetén referencia típus) Referencia értéket tárolunk. Pl. Egyenlőséget belső állapot alapján vizsgáljuk pl. String equals

2. Gyakori tervezési minták nagyvállalati gyakorlatban

Tervezési minta egy bevett gyakorlat, sok év programozói tapasztalat alapján kialakult jó szokások, bevett megoldások, minták. Adott problémát, mintát felismerve egy "best practise", hogyan kell ezt megoldani.

2.1. Létrehozási (creational) tervezési minták

Célok:

- Egységbe zárni azt az ismeretet ami a létrehozáshoz kell, pl. az objektum belső felépítését pl. más objektumokra hivatkozását szeretnénk egyetlen helyen kezelni. Később csak ezen az egy helyen kell majd módosítani.
- El akarjuk rejteni egy osztály konkrét implementációját.

Pl. Sok osztályban inkább csak egyetlen létrehozó osztálytól akarunk függeni, mint sok sok létrehozást duplikáltan létrehozni a sok osztályban.

Objektum vagy osztály alapú kategóriái vannak.

Builder pattern Az objektumot felépítését a reprezentációjától szeretnénk elkülöníteni. A builderben "beállító „ metódusok és egy "build „ metódus van. A builder Kompozícióval hozza létre a kívánt osztályt. Egy osztálynak lehet több buildere ezeknek kell egy közös builder ősük.

Factory method Készítsünk egy létrehozó osztály interfacet és az ebből leszármaztatott osztályok döntsék el hogy milyen konkrét példányt készítenek. Virtuális konstruktor. Van egy osztály legyen "F„ ami (egy metódusa) létrehoz egy objektumot, de ez a metódus visszatérésként absztrakt osztályt interfacet ad vissza, de "F„ osztály leszármazottai ezt felül definiálják és más más konkrét implementációval térnek vissza.

Static factory method Az osztálynak csak privát konstruktora van, de van egy statikus pl. "newInstance" metódusa amivel létre tud hozni magából egy példányt. pl. Calendar osztály. pl. Ha új szálat akarunk indítani egy konstruktorban ami nem jó, mert akkor még nincsenek a mezők inicializálva, ehelyett, ezzel a mintával ezt megoldhatjuk.

Simgleton pattern Azt szeretnénk, hogy egy osztályból egyetlen példány készüljön, de ehhez szeretnénk globális hozzáférést biztosítani. Statikus létrehozó metódus mindig ugyanazzal a példánnyal tér vissza. Többszálú környezetben figyelni kell a konkurens hozzáférés helyes megvalósítására. Általában előbb-utóbb több példányra lesz majd szükség, ezért érdemes ezt olyan keretrendszerre bízni ahol könnyen leváltható ennek a mintánk a használata.

Abstarct Factory Factoryt gyártó factory, termékcsaládok legyártására.

Prototype pattern Felkonfigurált objektumot másolással bővítjük (tipikusan JS) Depp copy = rekurzívan minden kompozit osztályt adattagjaival együtt lemásolunk. Shallow copy = a hivatkozott osztálynak csak a hivatkozását másoljuk így a másolat és az eredeti osztály is ugyanarra az osztályra fog hivatkozni.

2.2. Szerkezeti (structural) tervezési minták

Hogyan érdemes strukturálni az objektumainkat. Kisebb objektumokat szeretnénk egy komplexebb feladat elvégzésére össze strukturálni.
Cél:

- Flexibilis kód.
- Változásra való felkészülés.

Ezek a minták jellemzően delegációra és kompozícióra építkeznek.

Adapter pattern Az adapter kényszerhelyzetet orvosol. Pl egy interface és egy implementációt nem illik össze akkor közéjük rakunk egy adaptert. Meglévő meg-

oldást adaptálunk saját használatra. Az adapter általában compozit kapcsolattal hivatkozik az adoptálandó objektumra.

Bridge pattern Válasszuk szét az absztrakciót az implementációtól. Az ötlet hogy öröklés helyett kompozícióval hivatkozzunk az implementációra. Futásidőben lehet cserélni az implementációt. Absztrakció csak továbbít. Az öröklés hátrányaitól megszabadulunk. Ha egy környezetben függetlenül módosul az absztrakció pl. a hogy mit várnak el a kliensek, és a az implementáció, hogy ezt hogyan valósítjuk meg akkor érdemes használni. pl. sql.DriverManager

Decorator pattern Plusz felelősséggel szeretnék dinamikusan bővíteni az osztályt. pl. streamek egymásba ágyazása.

Facade pattern Egységes, egyszerű hozzáférést biztosít egy bonyolult alrendszerhez. Egy interfce, több interface számára egy alrendszerben. Magasabb szintű hozzáféréssel egyszerűbbé teszi az alrendszer használatát. pl. JPA, rétegek közötti kommunikáció könnyítésére használják.

Proxy pattern Helyettest biztosítunk egy objektumra. Pl. Hozzáférés kontrollálására használjuk. Közös interface. Általában csak továbbít plusz funkciót nem végez. pl. Lazy loading.

Composite pattern Rész egész viszony leképezésére van pl. xml DOM. rekurzív használata.

Flyweight pattern Optimalizációs céllal jött létre. Ha egy objektumra gyakran szükség van, és sokat nyerhetünk azzal ha több kliens is ugyanazt az objektumot használja.

2.3. Viselkedési (behavioural) tervezési minták

Objektumok közötti kommunikációt és az összetett felelősség felosztását szeretnénk megtenni. Laza kapcsolatot szeretnénk, ezért főleg kompozíciót fogunk használni.

Command pattern Egy kérést ne metódus hívás jelképezzen, hanem zárjuk a kérést egy objektumba. Szétválaszthatjuk a hívás és a végrehajtást. Fontossági sorrendet alakíthatunk ki. Kérés sorokat hozhatunk létre (batch) és visszavonási műveletet valósíthatunk meg.

Iterator pattern Kollekció elemeihez szeretnénk úgy hozzáférést biztosítani, hogy ne kelljen felfedni a kollekció belső szerkezetét. External: Iterátor készül egy kollekcióhoz. Az iterátor belelát az kollekcióba, általában belső osztály. Pl hashmap esetén nem lehet elkerülni a használatát.

Mediator pattern Sok-sok objektum kommunikál egymással valamilyen gráffal ez leírható. Ahelyett hogy, az összes objektum ismerje az összes másik objektumot, jól jöhet egy mediátor, közvetítő aki a objektumok közti kommunikációért felelős. A mediátort mindenki ismeri, a mediátor pedig ismeri az összes kommunikálni kívánó osztályt. És az objektumok ha egymással akarnak kommunikálni akkor a mediátorhoz fordulnak.

Observer pattern Több-több kapcsolat objektumok közötti kommunikációban. Ha egy objektum állapota megváltozik akkor ez az objektum minden erre a változásra feliratozott objektumot értesít a változásról.

Strategy pattern Definiálunk egy algoritmus családot, különböző algoritmusokat egységbe zárjuk és így egymással felcserélhetővé tesszük az algoritmusokat. pl. Layout manager. Közös interface az összes támogatott algoritmusoknak. Egy kontextusból tudja az algoritmus a számára szükséges adatokat megszerezni. Nagyon hasonló osztályok esetén, amiknek a viselkedése csak kicsit különbözik, akkor érdemes használni, kódismétlés elkerülésre.

Template method Vázat adunk egy algoritmusra és a leszármazottak, és a leszármazottak csak egy-két lépést felül definiálnak, de a vázat megtartják. Öröklést használunk.
Pl. servlet.
Final matódus legyen amit nem lehet felüldefiniálni, de amit kell azt absztrakt metódusba rakjuk.

3. Verziókezelési stratégiák, git flow

3.1. Mik a verziókezelés előnyei?

Változtatások nyilvántartása: ki, mikor, mit változtatott. Korábbi állapot megtekintése, visszatérés rá.

3.2. Mit jelent az elosztott verziókezelés?

A lényeg, hogy nem egy központi helyen vezetjük adatok változtatását. Változtatások lokális kezelése mások megzavarása nélkül. Egy gépen is megtalálható a teljes változás követés összes változata.

3.3. Milyen Git parancsokat ismersz?

- config
- init
- clone
- status
- add
- commit
- push/pull
- branch
- checkout
- merge
- rebase
- reset

3.4. gitk, .gitignore mire való?

gitk Grafikus program githez ami a commitokat és brancheket tudja megjeleníteni.

.gitignore Ebben a fájlban felsorolhatjuk a verziókezelésből ignorálni kívánt fájlok nevét. Csak arra a mappába és annak almappáira vonatkozik ahol létrehozzuk.

3.5. Mit jelent a távoli repository?

Az a kitüntetett repo amihez hozzá tud férni több munkatárs is arra használják, hogy lokális módosításaikat szinkronizálják. Általában szolgáltató adja pl. Github, Gitlab, BinBucket

Hogyan lehet kommunikálni vele? git remote add távoli repository hozzáadása lokális repohoz, cloneozáskor megtörténik automatikusan. majd pl. a lokális master branch trackeli a remote master branchet. Pull és push segítségével lehet műveleteket végezni, kommunikálni.

3.6. Mi az a Pull/Merge request?

Mind kettő jelentése: Egy fork branch vissza mergelése az eredeti repoba.

Nem a git , hanem a git menedzselő alkalmazás része ez a funkció. GitHub: pull request, GitLab: merge request. Csak a nevük különbözik.

3.7. Mi a különbség a merge és a rebase között?

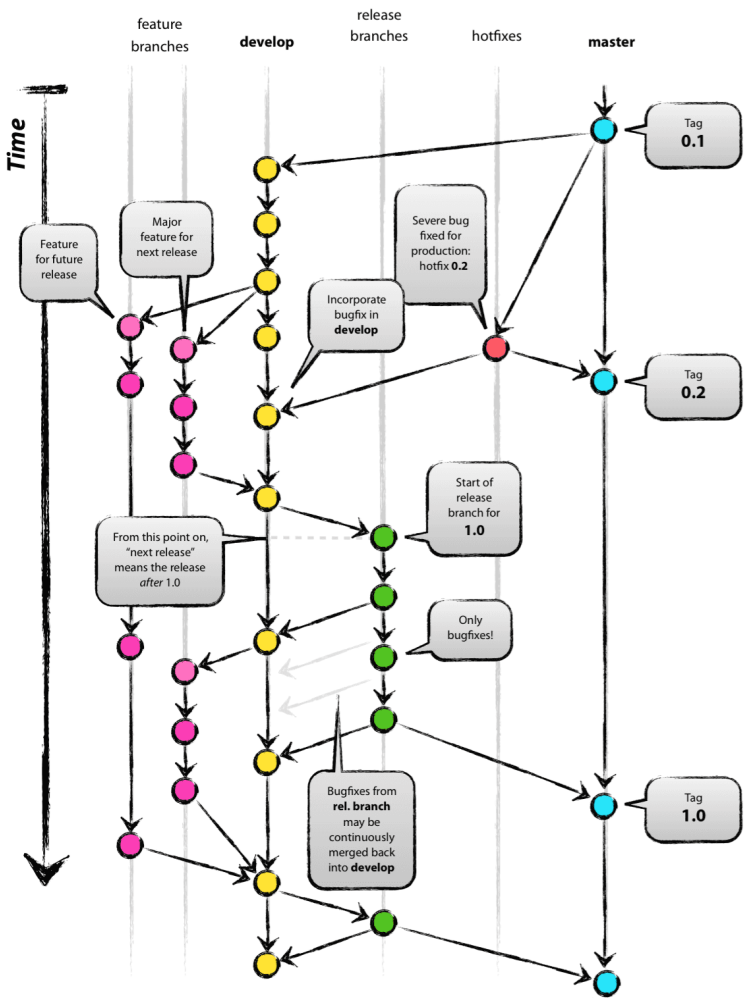
Merge Két ág összefűzése, a két ág külön külön megmarad és egy merge committal kerül egybe.

Rebase Egy ág összes commitját átmozgatjuk egy új ágba.

Mikor melyiket érdemes használni? Erről vallásháborúk vannak. A rebase akkor jó, ha lineáris commit historyt szeretnénk. Általában ha egy kisebb 1-2 fős csapat dolgozik, egy feature branchen akkor ott rebase-t használhatnak, **de közös megosztott branchen nagyon nem ajánlott a használata.** Merge együtt tartja az összetartozó komitokat, de az linearitás sérül, én merge párti vagyok.

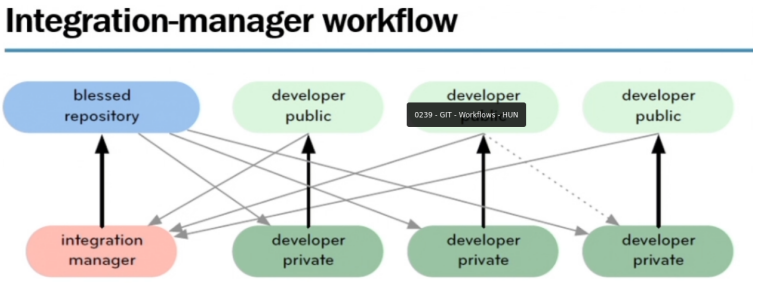
3.8. Ismerted a tréning anyagban szereplő branching modellt

master, develop, feature, release, hotfix

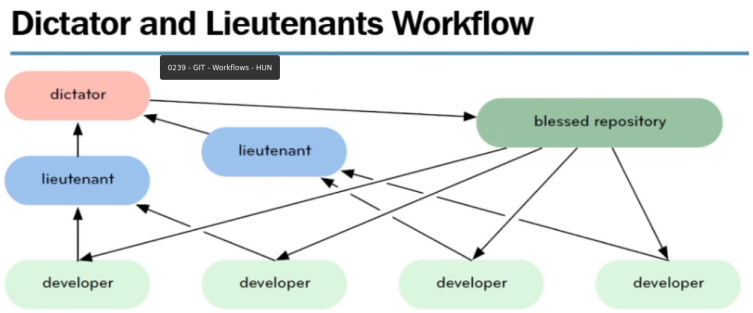


3.9. Milyen Git workflow-kat ismersz?

– Egyszerű egy közös központi repoba pusholás.
– Integration-manager workflow



Integration manager-en keresztül lehet a fő repot módosítani.
– Diktátor workflow



4. XML és JSON

Dokumentum leíró nyelvek.

4.1. XML fogalma

eXtensible Markaup Language

4.2. Mit jelentenek a következő fogalmak?

XML TAG Egy elem a fa szerkezetben. pl.

```
<button>b</button>
```

Attribútum Egy elemnek lehet tulajdonsága. De a gyerekelem jobban bővíthető, módosítható. pl.

```
<button attr="asd">b</button>
```

Jól formázott XML (well formed

- Nyitótag lezárása kötelező.
- Elemek lezárásának sorrendje a nyitás sorrendjével (fordítottan) egyezzen, ne legyen overlap.
- Egy gyökér elem legyen.
- Attribútumok értékek legyenek kódolva. Védett karakterek.
- Egy attribútum kétszer ne legyen megadva.
- Komment és tag nem keresztezheti egymást.

XSD XML Séma Definíció
Egy leírás tervrajz arra hogy hogyan kell kinéznie, milyen tageket attribútumokat tartalmazhat egy dokumentum ami a adott sémát megvalósítja. A séma készítéshez is van egy séma.

Valid XML Egy adott sémának való megfelelés.

4.3. Milyen XML feldolgozási módokat ismersz?

JaxP java API for XML processing

DOM parsing Egész XML dokumentumot a memóriában kezeli, nagyon kényelmes, de sok memóriát használ. Véletlenszerű hozzáférés az összes elemhez, módosítani lehet a fát.

Push parsing SaX / JAXP Egyszer végig fut a dokumentumon és eseménykezelők definiálásával lehet kezelni. Read-only, lineáris bejárás, nincs random hozzáférés. Eseményeket kapunk a parsertől, hogy milyen elem jött, majd eldönthetjük, hogy azzal mit akarunk csinálni.

Pull parsing StaX Hasonló események mint Push esetén. Plusz cursorral iterátorral lehet mászkálni. Read-only, lineáris bejárás, nincs random hozzáférés. Ezek eddig túl alacsony-szintűek.

4.4. Mi a JaxB?

Java API for XML binding. Kényelmes, ezt szeretjük használni. Marshal: Java object to XML Unmarshal: XML to Java object. Kell xml kell séma és fel kell annotálnia az osztályunkat.
xjc sémából osztály generál annotálva.

5. Build eszközök, Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

5.1. Mit jelent a build automatizálás?

A buildelést nem manuálisan végezzük, hanem van erre egy program, szkript vagy eszköz amit elindítva megcsinálja ezt automatikusan.

Miért hasznos? A manuális buildből származó emberi hibákat letudjuk csökkenteni. Bonyolult bild folyamatok kód generálást is használnak pl DAO, ezekhez konfigurációkat interfaceket használnak, ezeket becsomagolják, ez rengetek munka lenne kézzel. Ennek a segítségével megvalósítható a continuous integration (CI). Ha új kódot írunk egyből ellenőrizhetjük CI segítségével, hogy minden okés-e, és nem kell megvárni a release-t.

5.2. Mire való Maven esetén a pom.xml fájl?

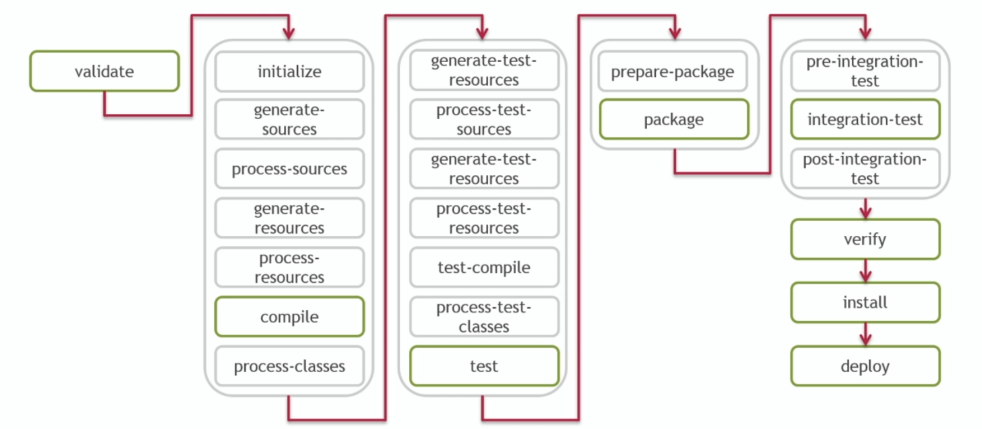
Egy projekt teljes leírását tartalmazza. Projekt Objektum Mapping.

Mit írunk bele?

- Verzió
- Group id
- Artifact id
- Java könyvtár függőségek.
- Modul függőségek.
- Esetleges ős modulból való leszármazás.
- Repository kezelés

Mit jelent a convention over configuration? Egy maven config alapértelmezetten a már mindenki által ismert, kialakult szokás alapján van beállítva és csak az ettől való eltérést kell megadni deklaratív formában.

Mi az a fázis, plugin, goal, lifecycle? A Maven életciklusa a következőképp írható le:
Vannak nagy fázisok. A fontosabbak: validáció -> compile -> test -> package -> integration-test -> verify -> install -> deploy Pl. A compile fázisban a compiler plug-int használjuk. A compiler plug-innak vannak góljai. pl. compile goal vagy test-Compile goal.



6. Java keretrendszerek, a Spring alapjai

6.1. Mit jelent a dependecy injection (DI)?

"Függőség beinjektálás." Keretrendszer végzi. A DI a IoC egy konkrét megvalósítása.

Milyen előnyei vannak?

- A függőség létrehozásának felelősségétől megkíméljük a DI-t használó osztályt.
- A létrehozás egy helyen van leírva, így könnyen módosítható később, mert nincs kód duplikáció.
- Singleton minta könnyen használható.

Spring esetén milyen fajtáit ismered?

- Konstruktor paramétereket,
- Settert,
- vagy mezőt is lehet injektálni, de nem javasolt.

6.2. Mi a Spring bean?

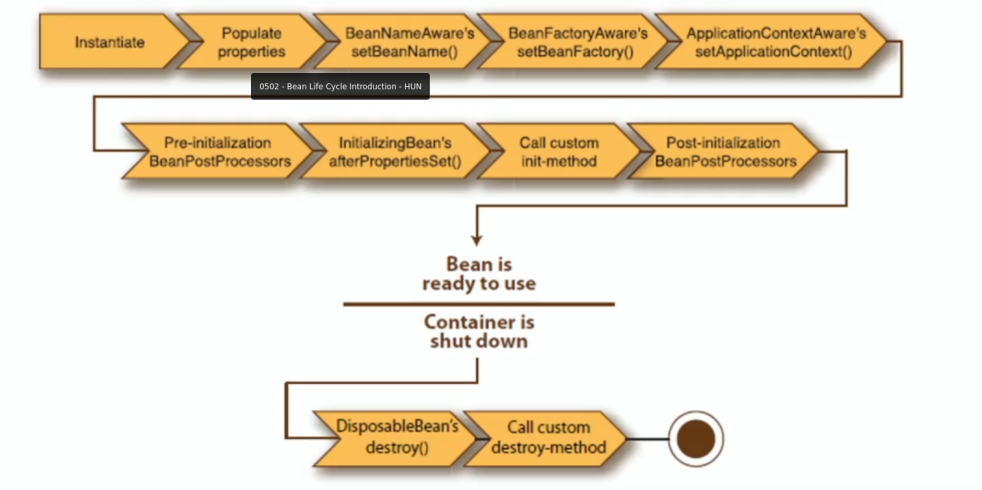
Olyan java objektum aminek a létezéséről a Spring tud, és Spring IoC konténer kezeli (pl. létrehozza, stb.).

6.3. Mi a Spring bean definíció (config)?

Egy leírás egy konkrét bean objektumra, ami alapján a Spring létre tudja hozni ha szükséges konkrét beaneket objektumokat. Függőségeket is itt lehet megadni.

6.4. Mi a Spring bean-ek életciklusa?

- Inicializáció
- Használat
- Megszüntetés



PI.
Singleton (alapértelmezett): A beanből egy példány jön létre.
Prototype: Mindig amikor szükséges újat hoz létre a bean definició alapján.

7. Spring keretrendszer részletei, spring framework

7.1. Mit jelent Spring esetén autowire?

Egy annotáció, aminek a segítségével tudunk függőséget beinjektálni oda ahova a kitesszük. Bean (config) definícióban lehet használni.

7.2. Mi az a BeanFactoryPostProcessor, Factory Bean?

PostProcesszor: lefut a bean életciklus lépései között, ennek a segítségével hozzá lehet adni kiegészítő kódokat a bean életciklushoz. Ezzel lehet Beant proxyzni, wrappelni kiegészíteni, dekorálni. Factory Bean: Spring megvalósítása a factory method viselkedésnek.

7.3. Milyen konfigurációs módokat ismersz?

XML Egy séma alapján lehet XMLben Beaneket definiálni.

Java Java osztály @Configuration nnotációval ellátva @Bean annotált metódusokkal lehet Beaneket létrehozni.

Annotáció DI is megvalósítható a komponenseken, belül annotációval így nem kell elkülönített config fájl ennek kezelésére.

Melyiknek mik az előnyei, hátrányai? Annotációval nem lehet mindent megcsinálni kicsit szétszórtabbá teszi a konfigurációt, de így is sokkal kényelmesebb mint az XML. Java konfig is sokkal kényelmesebb mint az XML, és mindent meg lehet vele csinálni. XML alapú konfig, régi módszer,mindent meg lehet vele csinálni ,fordítás nélkül lehet módosítani ennyi az egyetlen előnye.

8. Adatelérési eszközök (JDBC)

8.1. Mire való a JDBC API?

Különböző adatbázisokhoz való csatlakozásra, adatküldésre és fogadásra használjuk. DriverManager kiválasztja a megfelelő drivert a választott adatbázishoz (url alapján), és létrejön a kapcsolat. Csinálunk egy statementet amin tudunk queryt futtatni. A query futtatásának eredménye ResultSet lesz.

Miért nem kényelmes a használata? Hibakezelés macerás: vendor specifikus hibakódokat ad csak vissza. CheckException, Resource bezárás kell.

Milyen kiegészítést ad a Spring a használatához?

- DAO support: különböző megvalósítások egységes kezelése.
- JdbcTemplate: DAO support JDBC fölé csak a lényeges queryt kell megírni a többit elintézi.
- Egységes exception hierarchia, csak runtime exception.

9. Fejlett adatelérési eszközök (jpa, spring data)

9.1. Mire való a JPA?

JPA egy szabvány amit lehet implementálni. Pl. konkrét implementációja a hibernate. Java persistent API
Cél: objektumok és relációik perzisztenssé tétele az adatbázisba.

9.2. Mit jelent a configuration by exception?

Egy csomó alapértelmezett beállítás van pl. tábla neve az osztály neve lesz stb. és csak ezektől való eltérést kell konfigurálni.

9.3. Milyen főbb annotációkat ismersz?

@Entity Annotált osztály entitás lesz.

@Id Annotált mező lesz a primary key.

@Column Annotált mező oszlop neve állítható.

@Basic LAZY Betöltés csak hivatkozás esetén.

@Enumeration Enumot el tud tárolni, string vagy szám alapján.

@Temporal Date esetén lehet időt vagy dátumot vagy timestampet tárolni.

9.4. Hogyan valósítható meg reláció a JPA segítségével?

Single-valued associations Én hivatkozok **egy** másik entitásra, ha csak én akkor @OneToOne , ha rajtam kívül más is @OneToMany annotáció kell.

Collection-values associations Ha nálam egy kollekció és azokra csak én hivatkozok akkor @ManyToOne ha azokra más is hivatkozhat akkor @ManyToMany annotáció kell.

9.5. Milyen módokon támogatja az öröklődést a JPA

Singe table Egy nagy táblában az összes leszármazott plusz egy mező, hogy milyen típusú az rekord.

Join table Összerakja külön táblákkal kibővíti az ősosztályt.

Table per class Ősosztály mezői plusz az új mezők külön táblákban.

9.6. Mi az a Spring Data

Még magasabb szintű adatbázis támogatás. pl. Spring Data JPA. Interface alapú programozás modell. Metódus név lapján generál implementációt. Deklaratív lekérdezés lefuttatás.

Milyen szolgáltatásokat ad a JPA-hoz képest? Repo build JPA alapján. Typesafe JPA lekérdezés. Transparent. Lekérdezés validáció bootstrap (spring épülés) közben.

10. Webfejlesztés – web alapok, servlet, jsp

10.1. Mi a különbség egy webszerver és egy alkamazásszerver között?

Webszerver kifejezetten HTTP tartalom kiszolgálására van. Alkalmazás szerver képes HTTP tartalmat kiszolgálni, de nincs megkötve csak erre pl. FTPt is kiszolgálhat.

10.2. Hogyan épül fel egy HTTP request és egy response?

- URL
- metódus típus: GET, POST, PUT, DELETE, stb
- Header: pl. content/type, agent, session stb.
- Paraméterek: get, post formában key-value párok.

10.3. Mire való a servlet és a JSP?

Servlet Egy API. Ennek egy implementációja pl. a tomcat, glassfish. Az az API ami a hálózattal való kapcsolattartást definiálja.
Servlet egy web komponens amit a szerverre deployolunk hogy dinamikus weboldalk készítsünk.

JSP JavaServer Pages. Egy HTML szerű leírása az oldalnak, dinamikus adatok által vezérelt alkalmazást lehet a segítségével készíteni. Servletre épül és servletre fordul.

10.4. Mire való a web.xml?

A szerver bejövő request kezelését lehet benne konfigurálni.

10.5. Mire jók a Scope-ok?(application, session, request)

request Beant készít egy request erejéig.

application Beant készít a webszerver kontextus életciklusába.

Session Beant készít ami a session létezéséig fog létezni.

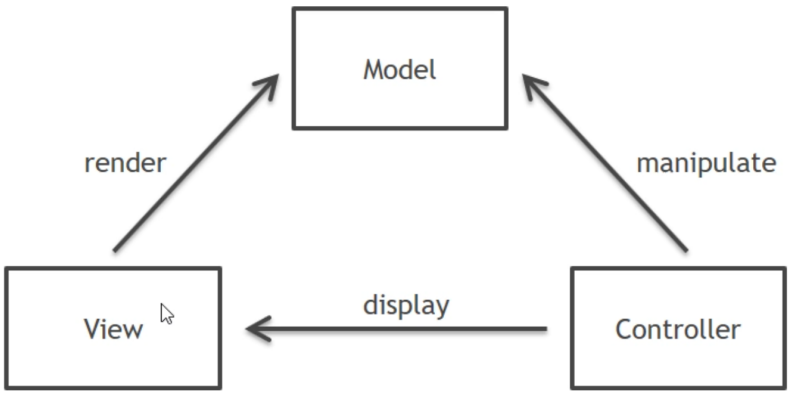
10.6. Mire jó egy cookie?

Kliens oldalon el tudunk benne tárolni kulcs-értékpárok formájában adatokat. Amik böngésző újraindítás esetén is megmaradnak. pl session id. tárolása.

11. Webfejlesztés - spring boot és spring mvc alapok

11.1. MVC fogalma

Model, View, Controller. Egy minta. Három aspektus, modul, réteg. View ismeri a modelt. A Controller is ismeri a modellt. A Conroller ismeri a viewt is.



11.2. Spring MVC project létrehozása Spring Boot segítségével

A spring boot egy eszköz ami arra lett kitalálva, hogy spring mvc framworköt használó alkalmazást lehessen készíteni gyorsan. Azonnali konfigurációkat tartalmaz, Spring alapú alkalmazás elindításához. stb.

11.3. @Controller feladata

Megkapja a requestet, készít egy modelt és egy megjelenítésnek tovább adja.

11.4. Tipikus MVC hívási lánc

JSP -> controller -> service -> repository -> domain

11.5. Security konfigurálása MVC segítségével (login form, WebSecurityConfigurerAdapter)

Egy Spring modul aminek a beiktatásával az oldalt elérését belépéshez köthetjük. Egy kijelölt Login és Register oldalak elvégzik az autentikációt. A Spring security pedig alapvető fontos biztonsági beállításokat, nem engedi az oldal elérést autentikáció nélkül.

11.6. Mi az a MockMVC?

MCV kérés-válaszokat lehet vele tesztelni. A tesztelés céljára konfigolja fel az MVC-t (nem kell teljesen).

12. Webfejlesztés –java alapú webes technológiák, spring mvc

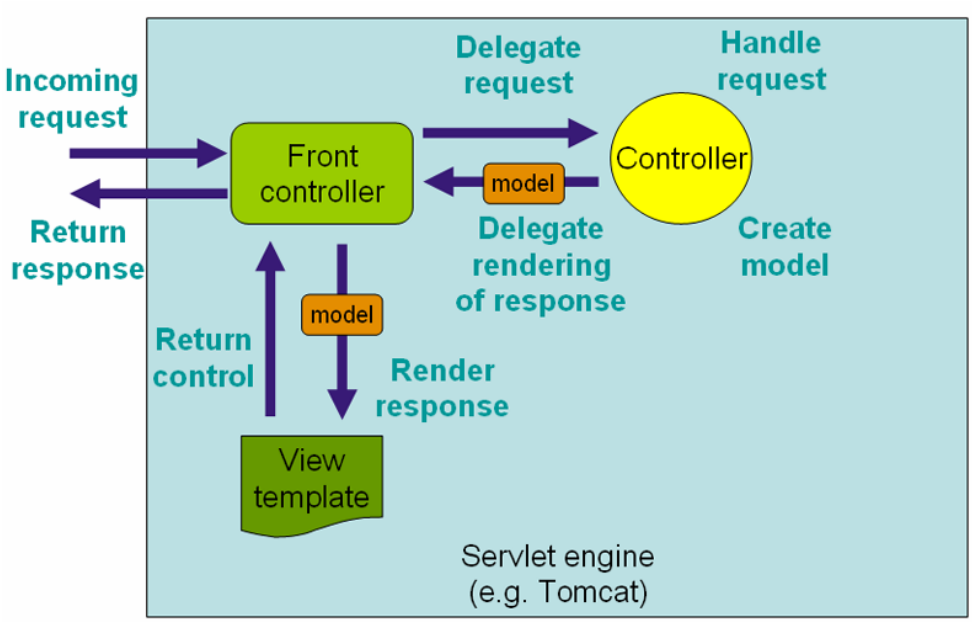
12.1. Többrétegű alkalmazás tipikus felépítése

- Web
  - JSP
  - Controller
  - Model
- Service
- Repository
- Domain

12.2. @Controller request mapping

Egy kontroller tipikusan egy request kezelésére van. A request mapping meghatározza, hogy milyen requesteket kezeljen a kontroller.

12.3. request processing lifecycle



12.4. Data binding

Adatkötés. request paramétereket név alapján domain objektumra tudja kötni.

12.5. Spring taglib

Egy JSPben használható könyvtár. pl. Két irányú adatkötést tud készíteni egy Spring form és data objektum között.

12.6. Form-ok létrehozása

Spring form tag használata után elérhető.

```
<% taglib prefix="form" uri="..." %>
<form:form ... >
```

12.7. CSRF védelem

Egy példa:
A böngészőben az egyik tabon be vagyunk jelentkezve a Bankba, és egy másik tabon mondjuk facebookon valaki készit egy gombot ami a háttérben a következőt linket nyitja meg: `bank.hu/utalas?ide=123&ennyit=9999` akkor mivel be vagyunk lépve csak egy másik tabon akár le is futhatna kérés, ez ellen akarunk védelmet, hogy a

mi alkalmazásunkat, még ha be is van lépve, ilyen linkekkel ne lehessen utasítgatni más oldalak által.

12.8. REST fogalma

Representational state transfer. Architektúra stílus. Egy köteg megszorítás egy webszolgáltatás elkészítésére, ha ezeket a megszorításokat betartjuk akkor az alkalmazásunkat APIt RESTful-nak nevezhetjük. HTTP protokollra épül. Alternatíva a SOAP-ra.

Megvalósítása Spring MVC segítségével @RestController annotáció segítségével, de a megszorításokat nekünk kell betartani.