Java alapú webes keretrendszerek

Java Enterprise Edition

Java platformok:

- Micro (J2ME/ Java ME): mobil eszközökre
- Standard (J2SE/ Java SE): desktop appok/appletek
- Enterprise (J2EE/ Java EE): elosztott, sok-felhasználós appok
    - a **Java EE** platform egy architektúra vállalati méretű alkalmazások fejlesztésére, a Java nyelv és internetes technológiák felhasználásával
        - vállalati információs rendszerek fejlesztése során **sok hasonló követelmény** (pl. biztonság, integrálhatóság más rendszerekkel)
        - ilyen feladatokat **érdemes általánosan megoldani**, **valamilyen keretrendszerre bízni**
        - a **Java EE a Java platform azon része**, mely **támogatást nyújt** ezen **igények megoldására**, azáltal, hogy különféle **szolgáltatásokat** nyújt, amelyek megkönnyítik a fejlesztést
            - többszálúság
            - tranzakciókezelés
            - biztonság
            - perzisztencia
            - névszolgáltatás
            - objektum életciklus kezelés
            - távoli metódushívás
            - aszinkron üzenetkezelés
            - skálázhatóság
            - terhelés-kiegyenlítés
    - **Java EE API-k:**
        - mindegyik **nyílt szabvány**, **több implementáció** lehetséges
        - Java Community Process (**JCP**) mechanizmus keretében, Java Specification Request (**JSR**) formájában definiáltak
        - JSR végleges verziójának kialakítása során az **ipari szereplők**, **fejlesztők igényeit** figyelembe tudják venni
        - Java EE API-kat megvalósító szoftvertermék az **alkalmazásszerver**
        - **mik ezek az API-k** most már:
            - Java Persistence API (**JPA**): objektum-relációs leképezés
            - Enterprise JavaBeans (**EJB**): elosztott, üzleti logikai komponensek
            - Java Message Service (**JMS**): aszinkron üzenetkezelés
            - Java Transaction API (**JTA**): tranzakciókezelés
            - Contexts and Depenency Injection for Java (**CDI**): függőséginjektálás
            - Java Authentication and Authorization Service (**JAAS**): biztonság
            - webes technológiák:
                - Java Servlet
                - JavaServer Pages
                - JavaServer Faces
            - webszolgáltatások:
                - Java API for XML-Based Web Services (**JAX-WS**): SOAP alapú XML webszolgáltatások
                - Java API for RESTful Web Services (**JAX-RS**): REST stílusú webszolgáltatások

- **tipikus Java EE alkalmazás architektúra**
    - többrétegű
        - **adat réteg:** feladata az adatok perzisztens tárolása, CRUD műveletek támogatása
          leggyakrabban relációs adatbázis
          tágabb értelemben ide tartozik minden olyan rendszer, amiből az alkalmazásunk adatokat nyer ki, ekkor szokás Enterprise Information System-nek nevezni (EIS)
        - **üzleti logikai réteg:** az a része az alkalmazásnak, amely a konkrét alkalmazási terület igényeinek megfelelő funkcionalitást biztosítja
          üzleti szabályok figyelembe vételével hívja meg az adat részeg szolgáltatásait
          Java esetén az üzleti logikai réteget EJB komponensekben valósíthatjuk meg!
        - **kliens réteg:** felhasználói felületet biztosítja, vagyis a felhasználói beavatkozások hatására meghívja a megfelelő üzleti logikai funkciót, majd megjeleníti annak eredményét
          2 típusa van:
            - **vastag kliens**: normál desktop app, leggyakrabban grafikus felülettel
            - **vékony kliens**: egy böngésző
        - **web réteg:** vékony kliensek csatlakozását teszi lehetővé
          az üzleti logikai réteg meghívása után rendszerint HTML választ generál a böngészők HTTP kéréseire
    - kliens-szerver
    - komponens-alapú
        - minden fejlesztett komponens egy **konténerben** fut, a konténer által nyújtott szolgáltatásokat éri el
- **Java EE tanúsítvány**: adott API-k megfelelő verziójának implementálása
    - nyílt szabvány → sok implementáció
    - Oracle adja, egy tesztsorozatnak kell megfelelni
- **Java EE Profile-ok**: Java EE 6 óta, célja a rugalmasabb technológia stack
    - lehetőség van arra, hogy egy szerver nem az összes technológiát (full profile), csak azok egy részhalmazát valósítja meg → **Profile**
        - ebben nem csak Java EE által megkövetelt technológiák szerepelhetnek, alakulhatnak a specifikációk üzemétől függetlenül is
        - Java EE előírja, hogy JCP definiálhat profile-t, megadja a módját, hogy kell hivatkozni a technológiákra
        - egy konkrét profile-t definiál is a Full Profile mellett: **Web Profile**
- **Pruning:** Java EE 6 óta, bizonyos régi technológiák eltávolítását jelenti
    - mint a deprecated metódus, csak technológiára mondja meg, hogy a köv. verzióban nem garantált a jelenléte, így nem javasolt a használata
    - Java EE 6-ban ilyenek pl.:
        - JAX-RPC, JAXR, EJB 2.0 és régebbi entity beanek, Java EE Application Deployment
- **elterjedt Java EE szerverek:**
    - Glassfish (refernecia-implementáció, open source)
    - IBM WebSphere Application Server
    - Oracle WebLogic Server
    - JBoss (most már Wildfly, open source)
    - Jetty, Apache Tomcat (ope source, csak webkonténer)
    - TomEE (Tomcat + OpenEJB)

Java Persistence API

Objektum-relációs leképezés:

- relációs tábla ↔ entitás osztály
- relációs tábla oszlopai ↔ entitás attribútumai (kell hozzájuk getter-setter)
- relációs tábla sorai ↔ entitás példányai

such műveletek:

- SQL SELECT → entitás megkeresése és betöltése a memóriába
- SQL UPDATE → entitás módosítása után az adat visszaírása a db-be
- SQL INSERT → entitás létrehozása
- SQL DELETE → entitás megszüntetése

ORM és EJB:

- EJB 2.1-ig az ORM az entity bean feladata volt – interfész és implementáció szétválasztása
- képes volt **automatikus O-R leképezésre** is, fejlesztőnek nem kellett JDBC kódot írnia
- memóriabeli bean példányok és az adatbázisbeli táblák sorai között a konténer szinkronizálta az adatokat
- **EJB 3** megtartotta a 2.1-es entity beaneket, de felvett egy új technológiát, a **JPA**-t
  - JPA entitás ≠ entity bean
  - JPA entitás = egy POJO perzisztencia megoldás, ami akár Java SE-ben is használható :3
  - API hívások során az app egy JPA implementációval (persistance providerrel) kommunikál, amely lecserélhető igazából (pl. Hibernate-re)
  - perzisztens modulként viselkedik minden olyan jar, amelynek META-INF könyvtárában van persistence.xml fájl, amiben definiáljuk a db JNDI nevét (javax.persistence csomag)

O-R leképezés annotációkkal:

- egy fájl `@Entity`-vel annotálva, argumentumok nélküli konstruktorral
- általában *implements Serializable*
- kötelező elsődleges kulcs: `@Id`, mellé mehet ID generáló stratégia: `@GeneratedValue(strat)`
- perzisztens attribútumok getterek/setterek formájában érhetőek el
- persistence provider elérheti közvetlenül a mezőket, ha azokat annotáljuk és nem a gettereket
  → JPA 2.0 óta osztályon belül is lehet variálni, pl. `@Access(FIELD)` / `@Access(PROPERTY)`
- többi annotáció már opcionális – táblák/attribútumok nevei automatikusan leképeződnek
  - `@Table(name=”ez_a_tabla_igazi_neve”)`, ha máshogy hívják az osztályodat
  - `@Column(name=”special_snowflake”)`, ugyanúgy, csak ezt attribútumra
- entitás attribútuma lehet egyébként:
  - primitív típus
  - String, BigInteger, BigDecimal, java.util.Date/Calendar, java.sql.Date/Time/Timestamp, byte[], Byte[], char[], Character[]
  - enum
  - más entitás (akár gyűjteményben)
  - nem-entitások gyűjteménye
  - beágyazott osztályok = olyan osztály, ami önmagában nem él perzisztens entitásként, csak egy perzisztens entitás példányhoz kapcsolódva
    - how to make one:

```
@Embeddable
public class SomeClass { java.util.Date aTablaEgyOszlopa; }
```

- how to use it:
```
@Embedded
@AttributeOverrides({
   @AttributeOverride(name="aTablaEgyOszlopa",
         column=@Column(„a_tabla_egy_oszlopanak_a_neve"))
})
public SomeClass getSomeClass() {...}
```

Perzisztenciakontextus:

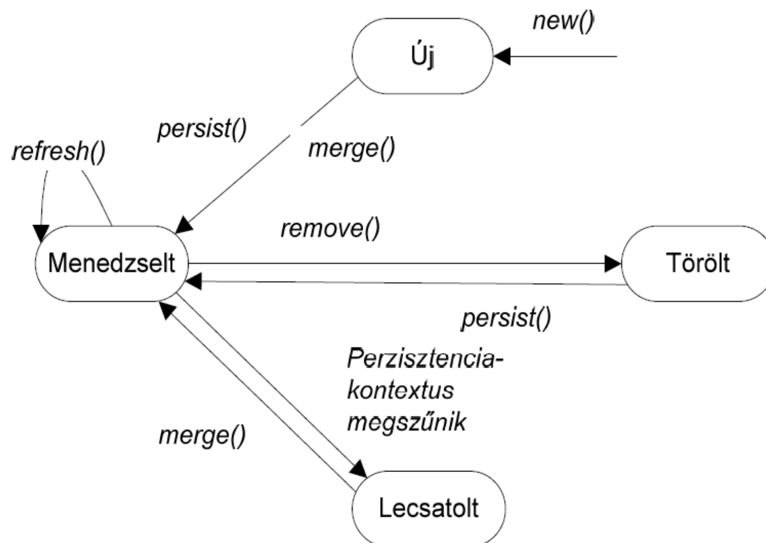
- = a perzisztencia provider által kezelt memóriában lévő entitások egy halmaza
- ezen keresztül kezeljük az entitásokat, ez a kapcsolat a memóriabeli entitások és a db közt
- API szinten az *EntityManager* interfészen keresztül kezeljük
  - 3 típusú metódus: entitások életciklusának kezelése
adatbázis szinkronizáció
entitások keresése
- egy EntityManager referencián keresztül egy perzisztenciakontextust érünk el, egy perzisztenciakontextuson belül minden entitás egy példányban fordulhat elő
- EntityManager életciklus kezelése kihívás → **függőséginjektálás** megoldja szépen
- ez akkor fog működni, ha a service-t olyan konténer hozza létre, ami függőséginjektálást biztosít → EJB konténer vagy Spring → **menedzselt perzisztenciakontextus:**
  - tranzakció elején jön létre
  - ha ugyanabban a tranzakcióban más osztályban is van injektált EM → ugyanazt a perzisztenciakontextust fogják látni
  - tranzakció végén záródik be (kivéve extended élettartamú p.kontextus esetén)
    - `@PersistenceContext` paraméterei:
      - `unitName`: ha több unit van a persistence.xml-ben, meg kell adni
      - `type`: az élettartamát határozza meg, csak EJB esetén releváns

entitások állapotai:

- **new**: new-val létrehozva kerül ide, csak a memóriában létezik, még nem íródik ki db-be
- **managed**: létezik az adatbázisban, és tartozik hozzá egy perzisztenciakontextus, amin ha meghívod a flush()-t, beíródnak a módosítások a db-be
- **detached**: adatbázisban megvan, de nincs hozzá perzisztenciakontextus → ebben az állapotban olyan, mint egy DTO, nem kell külön DTO osztály
- **removed**: még perzisztenciakontextus tartozik hozzá, de már ki van jelölve, hogy törölve lesz az adatbázisból

entitások életciklusa:

- új entitás menedzseltté tétele:
  - `persist()`: ha elsődleges kulcs ütközés van, kivétel
  - `merge()`: ha elsődleges kulcs ütközés van, SQL UPDATE, ha nincs, INSERT
- a `merge()` visszatérési értéke a menedzselt entitás példány!
- entitás lecsatolása:
  - perzisztenciakontextus kiürítésével: `em.clear();`
  - perzisztenciakontextus bezárásával: `em.close();`
  - JPA 2.0 óta akár egyedileg: `em.detach(entity);`



1. ábra, entitások életciklusa

- életciklus callbackek:
    - `@Pre/PostPersist, @Pre/PostRemove, @Pre/PostUpdate, @PostLoad`
    - perzisztencia provider hívja őket
    - ha külön osztályba akarjuk rakni, `@EntityListeners`-ben kötjük hozzá az entitáshoz az osztályt, és a metódusok az entitást kapják paraméterül

adatbázis szinkronizáció:

- a provider az EntityManager 2 metódusa segítségével szinkronizál az adatbázis felé/felől:
    - `flush()`: beírja a db-be a teljes perzisztenciakontextus összes módosítását
    - `refresh(entity)`: beolvassa db-ből a válotztatásokat egy entitásra
- Spring vagy EJB környezetben ritkán hívjuk ezeket, mert a tranzakció commit előtt automatikusan flush hívódik
- `EntityManager.setFlushMode()`:
    - `AUTO`-ra állítva (default) minden lekérdezés előtt hív egy flush-t, hogy a query a legfrissebb állapoton hajtódjon végre
    - `COMMIT`-ra állítva a tranzakció végén van csak flush
        - jobb teljesítmény
        - tranzakción belüli módosítások nem befolyásolják a lekérdezés eredményét (bad)

lekérdezések: mindegyik az EntityManager-en keresztül megy

- keresés elsődleges kulcs alapján:
    - `<T> T` **`find`**`(Class<T> entityClass, Object primaryKey)`
- lekérdezés teljesen dinamikusan:
    - JPQL nyelven: `public Query createQuery(String jpqlString)`
        - SQL-hez hasonló nyelv, de entitás példányokkal tér vissza
        - pl. `SELECT a from Account a WHERE a.balance=?1`
    - natív SQL: `public Query createNativeQuery(String sqlString)`
- statikusan definiált, névvel azonosítható lekérdezés:
    - `public Query createNamedQuery(String nameOfQuery)`
    - a lekérdezés `@NamedQueries`-ben van definiálva az entitás osztálynál
- Query:
    - **`setParameter`**: név vagy index alapján
    - **`getSigleResult()`**
    - **`getResultList()`**
    - **`executeUpdate()`**
- hasznos lehetőségek JPQL-ben:
    - többes törlés, módosítás (nem kell megtalálni a törlendőket, és egyesével törölni őket)
    - `JOIN`, `GROUP BY`, `HAVING`, subquery
    - paraméterek `?1`, `?2` helyett `:parameter_neve` formában (`?1` = első paraméter fv-ben)
    - projekció (csak bizonyos attribútumokat adjon vissza Object[] formájában)
    - új objektum létrehozása selectben:
      a visszaadott oszlopokat egyből valamilyen általunk definiált objektumként kapjuk (pl. findAll override, hogy `List<Entity>` legyen a visszatérési értéke)

Criteria API:

- JPA 2.0 vezette be, deklaratív, string alapú JPQL **objektumorientált**, **típusbiztos** alternatívája lekérdezések előállításához (több kód, mint JPQL esetén, de cserébe típusbiztos)
- Metamodell API segítségével elérhető string alapú navigáció
- a **perzisztencia egység metamodellje** metaadatokat tartalmaz az entitásokról és beágyazott osztályokról
- **általában annotációk feldolgozásával generálja a perzisztencia provider**, de kézzel is írható

- hogy néz ki ez miről van szó:

```
@Entity
public class Employee {
   @Id Long id;
   string firstName;
   string lastName;
   Department dept;
}
@StaticMetamodell(Employee.class)
public class Employee_ {
   public static volatile SingularAttribute<Employee, Long> id;
   public static volatile SingularAttribute<Employee, String> firstName;
   public static volatile SingularAttribute<Employee, String> lastName;
   public static volatile SingularAttribute<Employee, Department> dept;
}
```

- kanonikus metamodell jellemzői (ilyet illik írni):
  - osztály név után _
  - public static volatile attribútumok, SingularAttribute, Collection/List/Map/SetAttribute típusúak, megfelelő generikus típussal
  - attribútumok nevei azonosak
- metamodell segítségével teljesen típusbiztos lehet a lekérdezés
- előre generált metamodell osztályok nélkül is elérhető a típusbiztonság, mert dinamikusan használható a Metadata API

öröklés:

- lehetséges leképezési módok:
  - egy tábla egy osztályhierarchiához
  - külön tábla gyermekosztályonként
  - egy tábla egy konkrét osztályhoz
- fontosabb annotációk: `@Interitance`, `@DiscriminatorColumn`, `@DiscriminatorValue`
- egyéb öröklési lehetőségek:
  - entitás származhat nem entitásból
    - `@MappedSuperClass` annotáció az ősosztályra – nem lesz külön táblája egyik leképezési stratégia esetén sem, de az ottani attribútumok bekerülnek a db-be
  - nem entitás származhat entitásból
  - entitás lehet absztrakt
    - nem példányosodhat, de le lehet képezni táblára, le lehet kérdezni

kapcsolatok leképezése:

- 4-féle kardinalitás:
  - `@OneToOne`
  - `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`
- irány szerint:
  - egyirányú
  - kétirányú
    - kapcsolat mindkét végén lévő entitásnak lesz kapcsolatmenedzselő getter/setter metódusa
    - a két irányt a fejlesztő tartja konzisztensen!!
    - kétirányú OneToMany = kétirányú ManyToOne
- kapcsolatnak mindig egy tulajdonos oldala van
- beágyazott osztályok között is definiálható kapcsolat

- példa kapcsolat definiálására:
  - tulajdonos oldalon: `@ManyToOne @JoinColumn(name="idegen_kulcs")`
  - másik oldalon: `@OneToMany(mappedBy="idegen_kulcs_attr")`
  - +getterek, setterek
  - `@JoinColumn` mellett `@JoinTable` is használandó, ha külön tábla tartja nyilván a kapcsolatot (ManyToMany esetén általában)
  - `@ManyToOne` kötelezően tulajdonos oldal, mert nincs mappedBy paramétere
- **cascade**:
  - mind a 4 kapcsolatdefiniáló annotációhoz megadható egy cascade elem, pl.:
    `@OneToMany(cascade={CascadeType.PERSISG, CascadeType.MERGE})`
  - ennek lehetséges értékei a PERSIST, MERGE, REMOVE, REFRESH, ALL, és azt adja meg, milyen EntityManager műveletek hívódjanak meg a kapcsolódó entitásokra is
  - default nincs semmi
- **orphan removal**:
  - `@OneToOne` és `@OneToMany` kaphat `orphanRemoval` attribútumot, ez flush hívásakor két esetben törli a kapcsolódó nem szülő oldali objektumot:
    - ha töröljük a szülőt (cascade=REMOVE-val egyenértékű)
    - ha a kapcsolatot menedzselt állapotban megszüntetjük
      ha az elárvult entitást más szülőhöz rendeljük, nem definiált a működés
- **fetch:**
  - mind a 4 kapcsolatdefiniáló annotációhoz megadható egy fetch elem, pl.:
    - `@OneToMany(fetch=FetchType.LAZY)`
  - azt adja meg, hogy egy entitás betöltésekor betöltődjenek-e a kapcsolódó entitások is
  - **LAZY**: nem töltődnek be, csak ha hivatkozunk rájuk → nem foglal memóriát, csak ha tényleg muszáj, cserébe +1 lekérdezés
  - **EAGER**: default, betölt mindent → gyorsabb, de több memóriát foglal
  - finomhangolási lehetőségek:
    - legyen LAZY, de azokban a lekérdezésekben, ahol tudjuk, hogy szükség lesz a kapcsolódókra is, használjunk fetch joint az EJB-QL-ben!
  - oszlopokra is definiálható fetch, @LOB vagy @Basic paraméterként
  - **fetch-hez kapcsolódó problémák:**
    - ha lusta betöltés miatt még nincsenek betöltve egy entitás kapcsolódó entitásai, és ilyenkor lecsatolódik, a lecsatolt állapotban nem lesznek elérhetők ezek a kapcsolódó objektumok :(
      - megoldás: eager fetch, vagy lecsatolódás előtt a szükséges kapcsolatokra explicit getter hívás
    - a lecsatolt, kapcsolatoktól megfosztott, kapcsolattulajdonos példány mergelésekor a perzisztencia provider az adatbázisban is törölheti a kapcsolatot, ha cascade merge van beállítva D:
      - megoldás: a lecsatolt példány mergelése helyett id alapján keressünk, és másoljuk át a módosításokat kézzel
- **collection típusú mezők nem-entitás elemekkel**:
  - JPA 2.-tól egy entitás Collection attribútumában az elemek lehetnek nem-entitások is: alap típusok vagy beágyazott osztályok
  - külön táblába mennek a collection elemei, idegen kulcs oszlop hivatkozik a szülő entitásra, de a collection táblája nem önálló entitás, nincs pl. elsődleges kulcsa sem
  - `@ElementCollection` és `@CollectionTable`-vel szabható testre
  - Lafy fetch az alapaértelmezett

Spring

Spring alapelvei:

- célja az eredeti J2EE fejlesztési modellnél egyszerűbb megoldások szolgáltatása
- Java EE 5, 6 sok alapelvet innen vett át
- interfészek használata, OO design
- rétegzett felépítés, kiválaszthatók a használni kívánt modulok
- komponensek legyenek izoláltan tesztelhetők
- kivételek RuntimeException-ök
- alkalmazáshoz elég egy webkonténer (vagy Java SE)

mit ad nekünk ez a csoda:

- egy pehelysúlyú, nem-invazív konténert, amely segít az alkalmazás objektumainak konfigurálásában, „összedrótozásában" (Dependency Injection)
- tranzakciókezelés egységes absztrakciója, lecserélhető tranzakciómenedzser (lokális, JTA)
- egyszerűbb JDBC-kód megírását támogató segédosztályok
- különféle ORM eszközök egyszerűbb használata (JDO, Hibernate, Toplink, JPA)
- teljes AOP támogatás (pl. deklaratív tranzakciókezelés)
- MVC felépítést követő, több megjelenítési technológiát támogató webalkalmazás keretrendszer (Spring MVC)

függőséginjektálás előnyei:

- objektumok nem drótozzák be az általuk tipikusan tagváltozóként használt konkrét osztályokat
- injektor végzi a komplex objektumgráfok előállítását
- komplex inicializáló kód megspórolható
- különböző környezetekben, eltérő működést érhetünk el csak az injektor átkonfigolásával
- unit teszteknél mock objektumokkal helyettesíthetjük a tesztelendő objektum függőségeit

függőséginjektálás lényege:

- egy megoldás egy Factory vagy ServiceLocator alkalmazása: `.getInstance()`
  - hátránya: több kód (Factory megírása)
lehet, hogy nem típusbiztos
unit tesztelés nehézkes
- egy jobb megoldás az **`@Inject`** használata
  - ez annyit csinál, hogy az ezzel ellátott konstruktorhoz/setterhez/mezőhöz az injektor keres egy implementációt, és létrehoz egy példányt belőle

Spring Dependency Injection:

- Spring az első megoldás függőséginjektálásra (korábban Inversion of Control)
- konfiguráció evolúciója:
  1. XML fájl
  2. Annotációk + Java osztályok (JavaConfig)
  3. Spring Boot: automatikus default konfiguráió classpath alapján
     - testreszabható .properties vagy .yaml fájllal
     - teljesen felülírható JavaConfig-al
- régebben csak setter injektálást támogatott, ma már konstruktor és mező injektálást is :3
- **bean**: spring által kezelt objektum (függőséginjektálást + más szolgáltatásokat kaphat)
- minden modul erre épül, így azok viselkedését az általuk nyújtott beanekbe injektált más (beépített vagy saját) beanekkel tudjuk testreszabni
- az `ApplicationContext` interfészen keresztül érhető el a DI funkcionalitás → ennek létrehozásához legtöbbször nem kell explicit kódot írnunk

Spring DI annotációval:
- `@Autowired`-el kell megjelölni az injektálási pontokat (vagy `@Inject`-tel régebben)
- a használni kívánt osztályokra kell pár annotáció, pl. `@Component`, `@Service`, stb.

Spring DI JavaConfig:
- ha nem akarunk retek XML konfigurációkat írni, csinálhatunk egy ilyet:

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() { return new MyServiceImpl(); }
}
```

Spring beanek élettartama:
- 5 lehetséges érték állítható be `@Scope` annotációval vagy scope attribútummal bean tagben
- saját scope-ok is definálhatók
- alapértelmezetten minden bean **singleton** = `ApplicationContext`-en belül egy példány van
- további lehetőségek:
  - **prototype:** minden szükséges bean példány újonnan jön létre
    - singleton-ba prototype függőség injektálás csak egyszer történik meg
  - **request, session:** csak webes környezetben
    - egy problem: singleton egyszer jön létre, a függőségei is egyszer injektálódnak be, onnantól azt fogja használni
    - megoldás erre:
      - az egyszer beinjektált függőség ne maga a bean legyen, hanem egy proxy, ami delegálja a kéréseket a megfelelő scope-ban lévő bean-nek
        `@Scope(value=”session”, proxyMode = ScopedProxyMode.INTERFACES)`
  - **global session:** csak portlet környezetben

Adatelérés támogatása Springben – DAO támogatás:
- támogatás JDBC, JPA, JDO, Hibernate, iBatis egyszerűbb használatához
  - saját `RuntimeException`-ből származó Exception hierarchiával burkolja a kivételeket, ha a DAO osztályt `@Repository`-val annotáljuk
  - injektálható az `EntityManager`, Hibernate-es `SessionFactory`, `JDBCTemplate`
  - egységes tranzakciókezelési modell (lokális/globális tranzakciók, akár deklaratívan)
- egységes interfész XML binding technológiákhoz (JAXB, Castor, JiBX, XMLBeans, Xstream)

JDBCTemplate:
- `Connection` nyitása
- `Statement` létrehozása
- iteráció `ResultSet`-en
- kivételek kezelése
- `Connection`, `Statement`, `ResultSet` bezárása

Spring Data:
- külön modul az adatelérés támogatására
- segítségükkel az adatelérési kód nagy része megspórolható, **Repository** interfészekkel
  - saját entitásra specifikus: `interface MyRepo extends JpaRepository<Entity, Long>`
  - konfig: `@EnableJpaRepositories`, vagy `spring-boot-starter-data-jpa` classpath-ban
  - injektálás másik beanbe: `@Autowired private MyRepo repo;`
- tud még lapozás/rendezés támogatást: `PagingAndSortingRepository`
- automatikusan generált lekérdezések az interfészbe felvett metódusok nevei alapján
- JPA mellett más adatelérési technológiák is támogatottak (JDBC, MongoDB, Elastic, stb..)
- Repository közvetlen publikálása REST interfészen (spring-data-rest)

tranzakciókezelés célja (ACID):

- **atomicitás**: egy művelet minden része sikerüljön, vagy semmi sem
- **konzisztencia**: a rendszer egy konzisztens állapotból egy másikba kerüljön
- **izoláció**: konkurens kliensek ne lássák egymás be nem fejezett változtatásait
- **tartósság**: hardver- vagy hálózati hiba esetén sem vesznek el módosítások

problémák konkurens adatelérés esetén:

- piszkos olvasás (dirty read)
- megismételhetetlen olvasás (unrepeatable read)
- fantom olvasás (phantom read)

izolációs szintek:

- a db izolációs szint befolyásolja, hogy ezen problémák közül melyik léphet fel
- minél több problémát küszöböl ki, annál több zárat alkalmaz, rosszabb a teljesítménye
- **READ UNCOMITTED:** más által írt, de nem komittált adatot is lehet olvasni
- **READ COMITTED:** csak komittált adatot lehet olvasni, ez a **default**
- **REPEATABLE READ:** olvasás után tranzakción belül újraolvasva az adat nem változik
- **SERIALIZABLE:** konkurens tranzakciók sorosítva futnak

izolációs szint	dirty read	unrepeatable read	phantom read
READ UNCOMITTED	✓	✓	✓
READ COMITTED	×	✓	✓
REPEATABLE READ	×	×	✓
SERIALIZABLE	×	×	×

- izolációs szintek beállítása – JPA használata esetén a JDBC kód generált, így nem a fejlesztő nyitja a kapcsolatot!
  - JPA specifikáció nem követeli meg, hogy definiálni lehessen az izolációs szintet, hanem az adatbázis defaultra hagyatkozik
  - custom JpaDialect fejlesztésével megoldható

Java EE tranzakciók szereplői:

- **tranzakciós objektum (komponens)**:
  - olyan alkalmazás komponens, amely érintett egy tranzakcióban
- **tranzakciós erőforrás**:
  - írni, olvasni lehet (pl. db, üzenetsor)
- **erőforrás menedzser**:
  - amin keresztül elérjük az erőforrást (pl. JDBC driver db-hez)
- **tranzakció menedzser**:
  - elosztott (=több tranzakciós erőforrás között) tranzakció esetén mindenképpen kell
  - levezényli a teljes tranzakciót
  - Java EE esetében az alkalmazásszerver része
  - az erőforrás menedzsereket koordinálja, az X/Open XA protokollon keresztül, amely támogatja az elosztott tranzakciókat, kétfázisú commit segítségével
  - Java EE a JTA API-t definálja az elérésére

JPA entitások és tranzakciók:

- persistence.xml-ben megadható, hogy lokális, vagy JTA tranzakciókezelést választunk
- **lokális**: az EntityManager-től elkért EntityTransaction interfészen keresztül kell kezelni a tr-t
- **JTA**: egy elosztott tranzakciókat, 2-fázisú kommitot támogató tranzakciómenedzsert a JTA API-n keresztül megszólítva indítjuk a tranzakciókat, vagyis a tranzakció a perzisztencia providert kikerülve, közvetlenül a JDBC drivert szólítja meg
  - EJB környezetben tipikus, mert a konténerben ott a JTA, de Spring esetén is megoldható

- az entitások módosításával járó műveleteket (persist, merge, remove, refresh) csak tranzakción belül szabad meghívni, ha a perzisztenciakontextus tranzakció-élettartamú
- ha a lekérdezésekkel megtalált példányok tranzakción belül olvasódnak be → menedzselt állapotba kerülnek!

tranzakciókezelés Springben:
- egységes API a tranzakciókezelésre, ami mögött bekonfigolható a konkrét tranzakciómenedzser implementáció
  - működhet JDBC connection szinten
  - használhatja a JPA EntityTransaction-jét
  - elérhet elosztott tranzakciómenedzsert JTA API-n keresztül
  - használhatja a JDO API-t

tranzakciókezelés típusok:
- **programozott:** Spring által adott API-n keresztül kódból indítjuk/zárjuk a tranzakciót (ritkán használjuk)
- **deklaratív:** metódus szinten annotációkkal vagy XML-ben szabályozzuk a tranzakciók indítását / végét → metódusnál kisebb egységekről nem rendelkezünk
  - `@Transactional`, ami paraméterezhető
    - `rollbackFor`: milyen kivételek esetén legyen rollback (default, ha RuntimeException)
    - `timeout`
    - `isolation`: izolációs szint (de lehet, hogy az adott technológia nem támogatja pl. JPA)
    - `value`: tranzakciómenedzser bean azonosítója
    - `propagation`: mi történjen, ha tranzakcionális metódusból másik tranzakcionális metódust hívunk

Szervletek

webes alkalmazások:
- vékony kliens alkalmazások előnyei:
  - felhasználónak nem kell külön appot telepíteni, elég a böngésző
  - szerver oldalon egyszerűbb karban tartani az appot, egyszerre frissül az összes felhasználónál az új verzió
  - szerver terheléstől függően skálázható
- felhasználó statikus és dinamikus tartalmat kap eredményül
- HTTP: állapotmentes protokoll
- mostanában háromrétegű architektúra dominál:
  - adatelérés – Data Access Layer
  - üzleti logika – Business Logic Layer
  - megjelenítés – Presentation Layer
- ezek közt nincs mindig éles határ, meg aztán lehet őket tovább darabolni is, whatever

szervlet:
- Java Thread alapú, OO technológia kérés-válasz protokollok szerver-oldali kezelésére
- skálázható
- konténer-szolgáltatások (biztonság, hibakezelés, failover, session-failover)
- Java objektum, egy keretrendszerre és API-ra építve kibővíti a HTTP szerver funkcionalitását
- API a Java EE specifikáció része
- URL alapú címzését egy egyszerű architektúra gyorsan végzi el
- elkészített komponens alkalmazásszerverek között hordozható

JSP – JavaServer Pages:

- elválasztja az alkalmazáslogikát a megjelenítéstől
  - kimenete XML / HTML / XHTML / …
  - JavaBeanek széleskörű támogatása
  - jobb karbantarthatóság, újrahasznosíthatóság
- saját XML tagekkel, Java osztályokkal bővíthető
- szervlet technológiára épül, „kifordított szervlet”
- JSP oldal: szöveges dokumentum statikus és dinamikus elemekkel (HTML, kódrészek, …)

Szervlet:

- HTML kód Java-ban
- bármilyen adatfeldolgozás
- egyszerű webes kérésekhez
- nem az egyszerűen karbantartható kimenetet szolgálja

- paraméter-feldolgozás
- ha új fájlformátumot szeretnénk támogatni
- ábrák, grafikonok generálása
- kerüljük a HTML kódok használatát

JSP:

- Java-kód HTML-ben
- egyszerűbb formázás, bonyolultabb feldolgozás
- kód szervletté fordul

- HTML megjelenítés, formázás
- **általában együtt használjuk őket: MVC**
  - **szervlet: controller**
  - **JSP: view**
  - **keretrendszerek: pl. Struts**

Java EE webalkalmazás:

- egy telepíthető csomag, ilyen szexi dolgokkal a belsejében:
  - webkomponensek (szervlet, JSP)
  - statikus erőforrás-fájlok (pl. képek)
  - segédosztályok, osztálykönyvtárak (.jar)
  - telepítés-leíró (deployment-descriptor)
- megjelenés:
  - könyvtárak és fájlok hierarchiája, VAGY
  - az előbbi hierarchia ZIP fájlba tömörítve (.war)
  - része lehet az alkalmazásnak (.ear), de függetlenül is telepíthető

könyvtárstruktúra:

- fejlesztéshez: tartsuk a forráskódot az alkalmazástól függetlenül
  - létezik IDE támogatás
  - egyszerűbb iteratív fejlesztés
  - később nem probléma a „véletlenül” kikerült kód
- Maven:
  - /pom.xml – projekt build fájl
  - /src/main/java – java forráskódok
  - /src/main/resources – nem fordítandó, de classpathba csomagolandó fájlok
  - /src/main/webapp – jsp oldalak, statikus tartalom, ..
  - /src/test/java – tesztek java forrásai
  - /src/test/resources – tesztekhez kapcsolódó nem java fájlok
- WAR fájl: telepíthető csomag, ZIP fájl, távoli gépes telepítéskor használjuk
- / - gyökér, alatta tetszőleges könyvtárstruktúrában statikus tartalom, JSP
- /WEB-INF/: nem publikus könyvtár
  - konfigurációs fájlok, JSP tag library leírók, lefordított osztályok, osztálykönyvtárak
- Servlet 2.5-ben nem követelő web.xml, ha csak JSP-t használunk

fejlesztési lépések:

- statikus tartalom – kép, CSS, …
- komponensek leírása – szervlet, JSP, Helper, Delegate class
- fordítás – IDE, Ant, Maven
- csomagolás (build) – IDE, And, Maven, megfelelő könyvtárstruktúra vagy .war

URL leképzés:

- Java EE webkonténer HTTP szerverként működik:
  - HTTP kérések URL-jét dekódolja
  - statikus tartalmat statikusként szolgál ki
  - dinamikus tartalmat a konfigurációból kiolvasott szervletnek továbbít
    1. alkalmazás megállapítása
    2. szervlet kikeresése
  - context root:
    - egy appra jellemző egyedi elérési út
    - telepítésleíróban állítható, alapértelmezett a .war fájl neve
    - http://szerver.com/app/index.jsp
    - konténer a leghosszabb illeszkedő context path apphoz továbbítja a kérést
  - appon belül a szervletekhez különféle alias path rendelhető telepítésleíróban

telepítésleíró:

- standard telepítésleíró – /WEB-INF/ web.xml, case sensitive, sorrend se mindegy
- szerverspecifikus beállítások – /WEB-INF/sun-web.xml
- életciklus-kezelők
- kontextus paraméterek
- szervlet definíciók – init paraméterek, URL mapping
- szűrők
- error mapping
- biztonsági beállítások
- session élettartam
- referenciák – környezeti változók, erőforrások, EJB referenciák

szervlet request-response:

- szervlet: általános célú komponens tetszőleges protokollra, de ez általában HTTP
- folyamat:
  1. fogadja a kliens kérését
  2. kinyeri a megfelelő paramétereket
  3. alkalmazáslogika alapján valamilyen adat, tartalom keletkezik
  4. visszaküldi a választ a kliensnek
- **request:** a kliens által küldött teljes információ elérhető
  - ki küldte, milyen adatokat küldött?
  - milyen HTTP fejléceket küldött?
  - milyen locale kóddal küldte?
  - **ServletRequest** interfész elérést biztosít a köv. dolgokhoz:
    - protokoll információ (http, https)
    - adatfolyam típusa (content type)
    - adatfolyam (inputstream)
    - kliens adatok (locale, ip address)
    - kliens által küldött paraméterek
    - request scope objektumok

    - a request objektum áthalad a szervletek és szűrők által meghatározott láncon
    - HTTP GET és POST paramétereket ugyanúgy lehet elérni belőle
    - request scope objektumot a szervlet konténer is beállíthat
- **response:** visszaküldendő adatok összeállítása
    - szöveges és bináris is lehet
    - HTTP fejléc, HTTP cookie
    - cache lehetőség
    - **ServletResponse** interfész szervletből kimenő, klienshez küldendő adatokat tartalmaz
        - Output Stream vagy Writer elkérése
        - content type
        - kimenet bufferelésének állítása
        - locale információk beállítása
    - **HttpServletResponse**
        - HTTP response status code, cookie
        - általában HTTP GET és POST

kérések továbbítása:

- request továbbítás: (forward)
    - paraméterek, attribútumok megőrzésével
    - szerver oldalon
- response továbbítás: (redirect)
    - új paraméterezés, új attribútumok
    - kliens oldal hajtja végre

HTTP session:

- kérések sorozata közötti állapot megőrzés → de hiszen a HTTP állapotmentes!
    - HTTP cookie
    - URL rewrite
    - hidden FORM fields
- lejárat: session timeout (default 30 perc) vagy session.invalidate()
- fontos erőforrás, de azért gazdálkodjunk, mit teszünk bele, sok felhasználónál elszállhat
- vigyázat: a session objektum megosztott, több szálon is kezelhetik, elosztott környezetben csak szerializálható objektumokat tegyünk bele, egyébként elveszik :(

szervlet szűrők:

- cél: megfigyelni, módosítani, közbeavatkozni a kérés, illetve válasz alapján
- szűrőket láncba szervezhetjük
- használati esetek:
    - hozzáférés biztosítása, blokkolása
    - cache, tömörítés, logolás, titkosítás
    - tartalom transzformációja
- mit csinál egy jó szűrő:
    - headereket vizsgál
    - request objektumokat alakít át
    - meghívja a következő szűrőt
    - response objektumot alakít át
    - megszakítja a szűrő láncot, átirányítja a hívást
    - kivételt dob
- szűrők sorrendje = web.xml-ben a <filter> definíciók sorrendje, URL leképezéstől független
- az elsőt a konténer hívja, az utolsó a szervletet hívja

szervlet életciklusa:

- init() – létrehozás után
- destroy() – megsemmisítés előtt
- service() – abstract, protokollfüggő – doGet(), doPost(), doXXX()

szervlet élettartam:

- szervlet konténer lehetőséget ad arra, hogy a szervletek életciklusának különböző eseményeire egységesen reagáljunk – ilyen **események** pl.:
  - elindítás, leállás
  - scope attribútum megváltozása
  - session indítás, leállítás (invalidate())
- **listener objektum**: valamilyen életciklus-interfészt implementál
- **web-konténer**: minden listener osztályból létrehoz egy példányt, az összes eseményre az első request feldolgozása előtt beregisztrálja ezeket az objektumokat

többszálúság szervletekben:

- webkonténer több szálon fogadja a konkurens kéréseket
- alapértelmezett esetben a webkonténer egy példányt hoz létre minden szervlet osztályból, amelyet a kliensek több szálon érnek el

Servlet 3.0:

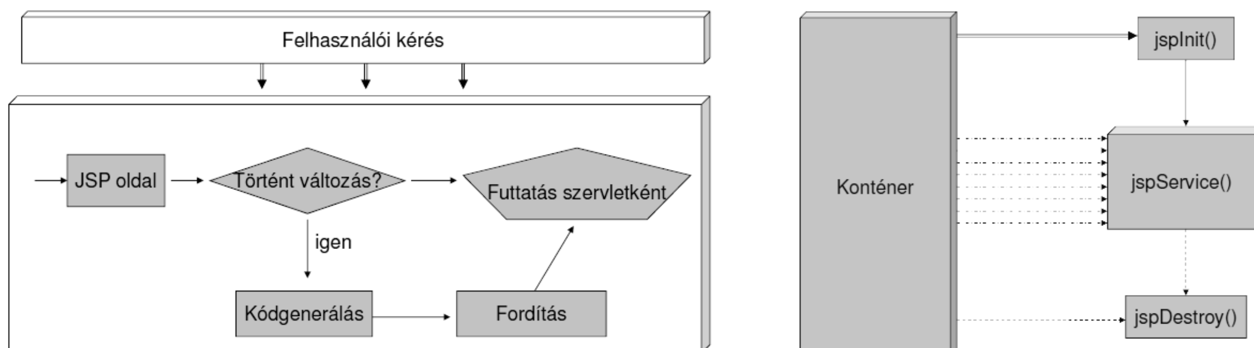
- webkomponens annotációk
  - cél: web.xml nélkül lehessen szervletet, filtert, listenert definiálni
  - `@WebServlet, @WebFilter, @WebListener`
- web fragments
  - webkonténer eddig a web.xml-ben kereste a konfigurációs információkat, de a web.xml most már innentől moduláris, szóval a WEB-INF/lib jar-jainak META-INF könyvtárában is keres a konténer web fragmenteket web-fragment.xml néven
  - tipikus felhasználás: webes framework-ök saját jarjaikban tudják az általuk igényelt konfigurációt elvégezni, nem kell a fejlesztőnek web.xml-t szerkesztgetni
  - le lehet ezeket tiltani, meg sorrendezni is lehet őket (abszolút, relatív), blablabla.
- webkomponensek regisztrálása futási időben
  - azon keretrendszerek konfigurálásának kiküszöbölését célozza, melyek alkalmazások között megosztottak (JAX-WS, JAX-RS, JSF implementáció)
- aszinkron feldolgozás
  - hosszan tartó kérések blokkolhatnak szálat, ez aszinkron kérésfeldolgozással kikerülhető

Servlet 3.1: nem-blokkoló IO

- HttpServletRequest olvasása és HttpServletResponse írása blokkoló volt,
- aszinkron feldolgozással ezt külön szálra lehetett pakolászni, mondjuk az is blokkolódott
- új interfészek: ReadListener, WriteListener
- új metódusok: ServletInputStream, ServletOutputStream
- csak aszinkron feldolgozás vagy upgrade esetén használhatók

Java Server Pages

JSP életciklus:



- a JSP oldal az app futása közben is módosítható
- JSP oldal megjelenítése előtt a konténer ellenőrzi, módosult-e legutóbbi fordítás óta
  - ha igen, akkor szervlet forráskódot generál belőle, majd Java compilerrel fordítja
- kliens kérése már ezen a szervleten megy keresztül
- 2-féle **szintaxis** van:
  - hagyományos – tömörebb, elterjedtebb
  - XML – JSP oldal érvényes XML dokumentum → meglévő XML eszközökkel feldolgozható
  - ezek vegyesen nem használhatók egy oldalon
- vannak **direktívák** is: `<%@ page/include/taglib %>`
  - import, contentType, szervletté fordulás előtt forrássá válik az includeolt fájl, …
- vannak **szkript elemek** is:
  - **`<%! deklaráció %>`** - itt deklarált változó a generált szervletben is deklaráció lesz
  - **`<%= kifejezés %>`** - tetszőleges Java utasítás, JSP végrehajtásakor kiértékelődik
  - **`<% kód %>`** - szkriptlet
    - felhasználhatók benne az oldalon deklarált változók
    - tartalmazhat lokális változókat, de
    - osztály/metódusdefiníciót nem, mert a szervlet doXXX() metódusába megy
    - eléri az implicit objektumokat (request, response, session, out, application, …)
    - statikus tartalommal keveredhet
    - használatuk kerülendő, le is lehet tiltani
- vannak **akció elemek:** hagyományos szintax is XML like
  - `<jsp:forward/include/plugin/useBean/getProperty/setProperty />`
  - JavaBean != Enterprise JavaBean, hanem egy olyan hagyományos Java objektum, ami mezői segítségével adatokat és műveleteket foglal össze – default ctor kötelező
- **tag library**: JSP oldalak között megosztható, újrafelhasználható komponens, saját akció elemeket definiál (a kis prefixről van szó a : előtt)
  - egyes tageket leíró fájlokból és az azok feldolgozását végző Java fájlokból áll, ált. .jar
  - egyedi URI tartozik hozzájuk, erre kell hivatkozni JSP oldalról
  - lehet csinálni **sajátot**, kibővítve ezzel a JSP lehetőségeit; attribútumokkal testre lehet szabni, visszatérési értékeket is adhat a hívó félnek, kommunikálhat más tag library-vel
  - milyen tag library-ket használhatunk:
    - Java Standard Tag Library (JSTL) – mezők elérése, iterációk, többnyelvűség, formázás, XML, db-elérés, String manipuláció, stb.
    - Jakarta-Taglibs, Struts tag library, Displaytag, ...

    - **tag fájl:** JSP 2.0 óta, tag handler alternatívája
        - segítségével JSP-hez hasonló szintaxisban definiálható a custom tag
        - .tag kiterjesztés, nem kell TLD
        - szép megoldás oldal sablonok kialakításához, az include bénácska
- dinamikus attribútumok, együttműködő tagek, környezeti változók és erőforrások elérése

JSP 2.0:

- újítások céljai: kevesebb Java kód JSP oldalon, „fejlesztőbarát” módosítások
- **scriptlet** – több lehetőség is van dinamikus tartalom megjelenítésére egy JSP oldalon, egyik legelső megoldás erre a `<%= kifejezés %>` volt, de most már van `${ kifejezés }` is, amit nagyon szépen úgy hívunk, hogy **expression language**
    - JSP konténer felismeri, mint szöveges sablont
    - programozhatóan is elérhető: javax.servlet.jsp.el csomagból
    - expression language egyedi függvényhívást is lehetővé tesz: `${ fn:barmi(akarmi) }`
    - JSP 2.1 óta **unified expression language**, ez lehetővé teszi a késleltetett kiértékelést

JSTL:

- hasznos custom tagek gyűjteménye
- open source implementáció
- expression language használható az attribútumok átadásakor
- korábban a JSTL tag library jarjait az apphoz kellett csomagolni
- Java EE 5 bevette a specifikációba = appszerver köteles implementációt biztosítani
- 5 csoportba oszthatók a benne lévő tagek:
    - **core:** prefix: c
        - *out*, *set*, *remove*, *if*, *choose-when-otherwise*, *forEach*, *import*, *url*, *param*, *redirect*
    - **függvények:** prefix: fn
        - *kollekciókra length*, *string manipulációs függvények*
    - **formázás:** prefix: fmt
        - *message*, *param*, *setLocale*, *formatNumber*, *formatDate*
    - **XML:** prefix: x
        - *parse*, *out*, *set*, *forEach*, *transform*
    - **SQL:** prefix: sql
        - *query* (*var*, *dataSource*)

displaytag lehetőségek:

- többnyelvű szövegek: Title helyett titleKey, táblázat fejléce, lábléce külső property fájlban
- lapozás
- rendezés oszlopokra: táblázat alapértelmezetten nem őrződik meg a session-ön belül, ilyenkor vagy nekünk kell session-be tenni, vagy újra le kell kérdezni
- szabadon állítható CSS
- XML, CSV, XLS, PDF export – szervlet filterrel valósítja meg

Webalkalmazás keretrendszerek

problémák komplexitása:

- **model 1**: csak JSP + JavaBeanek
  - többrétegű alkalmazás, egymásra épülő rétegekkel
  - webes réteg tisztán request-response alapú
  - mi problémák vannak: nehéz átlátni, módosítani és bővíteni
- **model 2**: Model-View-Controller
  - sokkal rugalmasabb architektúra, jól elkülöníthető feladatok és szerepek
  - három különböző feladat:
    - **model**: alkalmazás adatai (EJB: üzleti logika, view állapota: beanek a scope-ban)
    - **view**: adatok megjelenítése (JSP oldalak Java kód nélkül)
    - **controller**: felhasználói események feldolgozása (szervletek, html kiíratás nélkül)
      - webappok esetén a controller tipikusan ezeket csinálja:
        - felhasználó által küldött paramétereket feldolgozza
        - valamilyen üzleti logika meghívása
        - hívás eredményének elérhetővé tétele view számára
        - következő view kiválasztása és odanavigálás
      - ezek újrafelhasználható, rugalmasan konfigurálható megvalósítása, + még néhány feature támogatás = webapp. Keretrendszer

Spring Boot

általános jellemzők:

- cél: a Spring egyszerűbb használata
- webalkalmazások is önálló Java alkalmazásként futtathatók, beágyazott webkonténeren → nem szükséges külön deploy
- megfelelő jar függőségek révén automatikus default konfiguráció
- default-tól való eltérés beállítása
  - egyszerűbb esetben properties/yaml fájl, egyébként JavaConfig, XML-n nem kell írni

függőségek kezelése:

- pom.xml, `spring-boot-starter-parent` mint szülő
- `dependencies` tagben függőségek, verziót nem kell megadni

how to run shit:

- mvn spring-boot:run, spring-boot-starter-web függőséggel beágyazott webkonténer indul
- default 8080-s port, root context

kód struktúra:

- app osztályt célszerű egy „root” package-be tenni, többi osztály az alatti packagbe legyen:
  - model: entitások
  - service: üzleti logikai osztályok
  - controller: Spring MVC kontroller osztályok
- `@SpringBootApplication`: három annotációt fog össze:
  - `@EnableAutoConfiguration` → kijelöli a packaget, amiben keresgélni fog a Spring Boot
  - `@ComponentScan` → komponenseket honnan keressen
  - `@Configuration` → ha app osztályban írunk JavaConfigot

konfig osztályok: JavaConfig > XML

- szétoszthatjuk több `@Configuration` osztályba, amik érvényre jutnak, ha a root package alatt vannak, és az app osztályon van `@ComponentScan` vagy `@SBA` annotáció, VAGY explicit behúzzuk őket, hogy `@Import(MeConfig.class)`

autokonfiguráció:

- `@EnableAutoConfiguration` vagy `@SBA`, engedélyezi az autokonfigurációkat, default mindent
- érdemes használni, okos, tudja kb. mit akarsz konfigolni, nem fogja felülírni a dolgaidat
- autokonfig osztályok működése testre szabható propertykkel
  - application.properties vagy application.yml (utóbbi tömörebb)
  - ezeket a szarokat a Spring itt fogja keresni sorrendben:
    - aktuális könyvtár /config könyvtára
    - aktuális könyvtár
    - classpath /config könyvtára
    - classpath
- propertyk több helyről jöhetnek, pl. parancssori argumentumból, webes context-paramból
- retek propertyt a kódból így tudsz elérni: **`@Value(„${myprop}”)`**
- a `@ConfigurationProperties` annotáció `@Bean` metódusokon is alkalmazható, a legyártott bean propertyjeit fogja konfigolni a megfelelő prefixű propertykkel
  - ennek vannak előnyei:
    - egyszerre injektálhatunk több, akár hierarchikus propertyt
    - nem tudjuk elgépelni a `@Value`-nak adott értékeket
    - megengedőbb a property nevekkel, ._case_sensitivity kitérdekellllllll
    - a `spring-boot-configuration-processor` metaadatokat tud generálni osztályokból, így a konfig fájlok szerkesztésekor kódkiegészítés is működhet IDE-kben

profile-ok:

- cél: bizonyos környezetekben más beállításokkal fusson az alkalmazás
- hasonlít a maven profile-okhoz, de ott a build során adjuk meg őket, itt futtatáskor :3
- profile aktiválása: spring.profiles.active nevű propertyvel, vagy parancssori kapcsolóval (ha mindkét módon megadjuk, a parancssori felülírja a propertyben megadottat)
- konfigfájlra lehet aztán rakni `@Profile(„production”)` annotációt, és akkor az jut érvényre
  - `application-{profile}.properties/.yml`
  - profile aktiválásakor lép életbe, magasabb precedenciával, mint a profile-mentes verzió
  - ha több profile-t adunk meg, az utolsóhoz tartozó konfigfájlnak van a legnagyobb precedenciája

fontosabb autokonfigurációk:

- **Spring MVC**: spring-boot-starter-web függőség → Spring MVC autokonfig
  - default felülírása: **`@Configuration`** annotáció, **`WebMvcConfigurationAdapter`** leszármazott felülcsapott metódusaival
  - autokonfig helyett teljesen saját konfig: **`@EnableWebMvc`**
- **Thymeleaf**: spring-boot-starter-thymeleaf függőség → autokonfig
- **DataSource**: starter-jdbc vagy starter-data-jpa függőség
  - ha nincs saját datasource, megpróbál automatikusan definiálni egy defaultot
- **JPA**
- **Spring Data:** spring-boot-starter-data-jpa függőség → default Hibernate is jön vele
  - entitások és repository interfészek minden konfig nélkül használhatóak
  - nem kell persistence.xml
- **Spring Security**
  - default BASIC authentikációval véd egy csomó url-t, kivéve néhány statikus vackot (css)
  - in-memory `AuthenticationManager`, 1 user, random jelszóval, konzolról lopható, mi ez
  - security események publikálása bekapcsolva
  - HSTS, XSS, CSRF védelem bekapcsolva

spring security testre szabása:

- `@EnableGlobalMethodSecurity` annotáció explicit kitételével
- property alapú konfig a `SecurityProperties` osztály alapján
- konfig teljes lecserélése: `@EnableWebSecurity`, `WebSecurityConfigurerAdapter`

tesztelés: spring-boot-starter-test → több hasznos teszt libraryt behúz (junit, spring-test, mockito)

- sima springes teszteléshez hasonlóan `@RunWith(SpringRunner.class)`
- de nem kell `@ContextConfiguration` megnevezés a teszt kontextusának, van helyette `@SpringBootTest` annotáció :3
- `@AutoConfigureTestDatabase` → lecseréli a beállított datasourcet egy embeddedre `@DataJpaTest` pl.

AJAX

RIA – rich internet application:

- hagyományos HTML-nél gazdagabb, vastag kliens jellegű funkcionalitást kínáló webapp
- többféle módon is megvalósítható, böngésző plugin kell hozzá, pl. Flash, Silverlight, stb.
- **AJAX – asynchronous javascript and xml:**
  - csak javascript kell hozzá
  - lényege: egy bizonyos JS objektum segítségével aszinkron módon küldünk HTTP kérést a szerverhez, aminek a válaszát szintén JS segítségével fogadjuk, és az aktuális oldalnak egy részét módosítjuk a kapott adatok alapján
  - mindet az aszinkronitásnak köszönhetően a felhasználó számára észrevehetetlen
  - tipikus AJAX-os funkcionalitás:
    - automatikus szövegkiegészítés
    - validáció gépelés közben
    - szerver oldalon változott adatok automatikus frissítése
    - adatok on-demand lekérése
  - hogy működik ez a csoda:
    - valamilyen DHTML eseményre JS függvényt definiálunk (pl. egy gombra)
    - ebben a függvényben az XMLHttpRequest JS objektummal aszinkron HTTP kérést küldünk a szerver felé, a választ egy JS callback-ben kezeljük, frissítjük a DOM-ot :3

PRO:

- rich client – előtelepített kód nélkül
- jobb felhasználói élmény
  - gyorsabb szerver round-trip (nem az egész oldal töltődik le)
  - kényelmesebb felületek (drag n drop)
- kisebb hálózati terhelés

CON:

- back gomb probléma
- URL-ek így már nem tükrözik az állapot-váltást
- böngésző függőség
- lassú gépeken a javasript futtatása problémás lehet
- csak a hívó domain-hez küldhetők kérések
- lehet, hogy le van tiltva a böngészőben a JS!!!!
- fejlesztés keretrendszer nélkül nehézkes :(

javasript libraryk:

- külön böngésző plugin telepítése nélkül a javasript a legelterjedtebb eszköz, amivel interaktívabbá tudjuk tenni a webes felületet:
  - AJAX-nak is ez az alapja, de egyéb teljesen kliens oldali vezérlők is jellemzően JS alapúak
- JS fejlesztés főbb nehézségei:
  - böngészőfüggő
  - interpretált → lassú lehet, nincs szintaxisellenőrzés
  - dinamikus típusok → csak futás közben észrevehető hibák, IDE-támogatott refaktorálás

hiánya
    - nem osztály alapú OO támogatásra lett kitalálva, bár bele lehet csempészni
    - nehezen detektálható memory leakek, akár böngészőfüggő módon
    - debuggolás (ma már mondjuk triviális, böngészők F12-ben adják)
- következmény: csak minimális funkcionalitást akarjunk saját JS kódban megvalósítani, minden másra ott a mastercard... i mean, támaszkodjunk bevált JS librarykre :(
- a JS libraryk használata egyszerű, a HTML head részében hivatkozunk a szükséges JS fájlokra

jQuery:

- rakás függvény DOM manipulálásra, navigálásra, effektekre, eseménykezelők definiálására
- a jQuery globális függvényre épül, ami a $ operátorral érhető el
- a függvény kiterjeszti a selector-ra illeszkedő DOM elemet, amin így jQuery-s függvények hívhatók
- a selector szintaxisa lehet a CSS-el megegyező, de jQuery specifikus selectorok is vannak:
    - :checked, :empty, :gt(), :lt(), :nth-child(), …
- híres neves jQuery varázslat a **`$(document).ready(function() { … });`**
- AJAX-ra ezt tudja ajánlani: `$.post(„test.php”, { vmi jo json }, function(){}, „xml”);`
- ha válaszba JSON-t kérünk, van egy sokkal szebb mágia:

```
$.ajax({
   method: „POST”,
   url: „register.php”,
   data: { vmi jo json }
})
.done(function() {…})
.fail(function() {…})
.always(function() {…})
});
```

- a jQuery UI a jQueryre építve valósít meg vezérlőket, további effekteket, témákat!
    - widgetek: Accordion, Autocomplete, Button, Datepicker, Dialog, Progresspar, …
    - interakciók: drag n drop, resize, selectable, sortable
- **knockout:**
    - kétirányú adatkötés:
        - HTML elemek és a viewmodel objektumok összekötése
        - deklaratív módon
        - kétirányú → egyik módosítása maga után vonja a másik változását is
    - MVVM:
        - model: szerver oldali adatok
        - view: GUI (HTML)
        - view model: GUI-n látható adatok absztrakciója

magasabb szintű JS framework-ök:

- arra adnak keretet, hogy a teljes appot JS-ben írhassuk:
    - adatkötés
    - oldalsablonok támogatása
    - MVC/MVVM felépítés
    - tesztelhetőség
    - kommunikáció szerver oldallal
- Angular, React, Ember.js, Backbone.js, …
- gyakran egyéb library-kre épülnek, pl. jQuery, Knockout, stb.

Angular JS:
- ng-* direktívák (ng-app, ng-controller, ng-repeat)
- szerver oldali adatok elérése: erre való a `$http` service
- oldalak közti navigáció: `angular-route` modulra támaszkodva

Angular 2 (4, 5):
- JS → TS (fordításnál JS-re fordul)
  - típusosság, generikusok, lambdák, stb...
- controllerek helyett komponensek
- fejlettebb függőséginjektálás
- jobb mobil-támogatás

REST API Spring MVC-vel:
- controller metódusaira `@ResponseBody` annotáció → metódus visszatérési értéke body-ba
- handler metódus bemenő paramétere `@RequestBody` → kérés törzse injektálódik
- formátumok között a `@RequestMapping produces` és `consumes` paramétereivel válthatunk (default application/json)
- Spring 4 óta: `@RestController` → `@ResonseBody` elhagyható :3

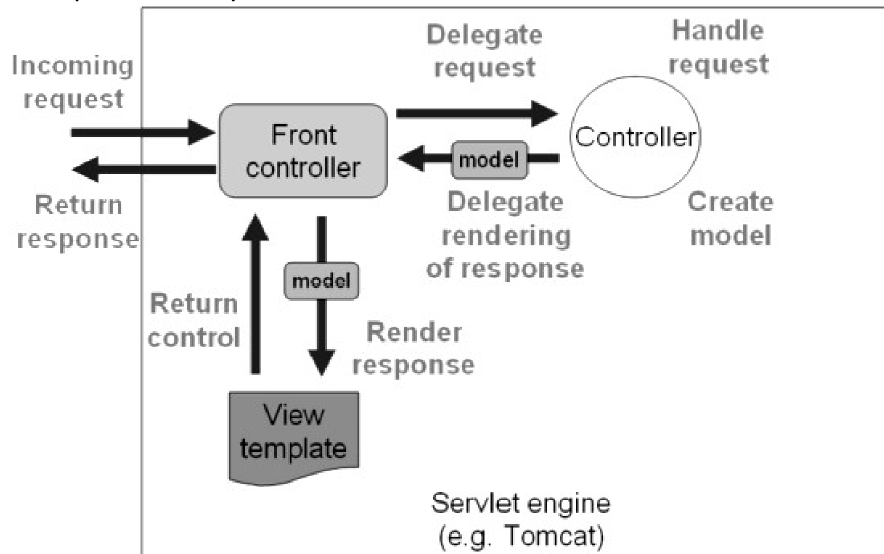
JSON szerializáció:
- default a Jackson libraryre építve, de testre szabható
- tipikus megoldandó problémák:
  - JPA entitások JSON-ben szereplő kapcsolatainak betöltése még menedzselt állapotban
  - adott kérésnél fölösleges vagy nem publikus mezők kihagyása JSON-ből
  - kapcsolatokon keresztüli körkörös hivatkozások kezelése (`@JsonIgnore`?)
- ezek kezelésére két lehetőség:
  - entitások helyett Data Transfer Onject-ek használata kontrollerekben
    - **PRO:**
      - megoldja a fentebb említett problémákat
      - REST API kliensei és a perzisztens model réteg között lazább csatolás
    - **CON:**
      - sok plusz osztály, amelyek gyakran az entitások másolatai → karbantarthatóság
      - DTO és entitás közti konverzió futási idejű overheadet jelent
      - DTO és entitás közti konverziót le kell fejleszteni
  - entitások JSON reprezentációjának testreszabása Jaskson vagy más library használatával
    - **PRO:**
      - megoldja a fenti problémákat
      - nincsenek plusz osztályok
      - nincs futási idejű overhead
    - **CON:**
      - annotációkkal telezsúfolt entitás osztályok
      - klienseknek követni kell a JPA model osztályok változásait

Spring MVC

- Springre épülő üzleti logika elé tetszőeges keretrendszerrel készíthetünk webréteget (JSF, Struts, Wicket, Portlet, stb.)
- de a Spring maga is kínál egy webalkalmazás keretrendszert = Spring MVC
- az oldalfolyamok kezelésére a Spring Web Flow szolgál, ez a Spring MVC-től függetlenül használható

Spring MVC általános jellemzői:

- objektumok felelősségeinek jó szétválasztása
- rugalmasan konfigurálható
- nincsenek előírt ősosztályok, interfészek
- annotációk támogatása
- testre szabható validáció
- több megjelenítési technológia támogatása → támogatás a modell átadása
- adatkötés és témák támogatása tag libraryvel
- nincs UI komponens modellje
- integrálható Struts-al, JSF-el, Tapestryvel, WebWork-el
- használható portlet környezetben is



controller osztály:

- bemenő paraméterek lehetnek:
    - ServletRequest, ServletResponse, HttpSession
    - WebRequest, Locale, Input/OutputStreamReader/Writer
    - @PathVariable, @RequestParam
    - @RequestHeader, @CookieValue
    - @RequestBody, **HttpEntity<?>, HttpMessageConverterrel**
    - java.util.Map, ModelMap, Model – aktuális modellre
    - @ModelAttribute-tal annotált modell-beli objektumok
    - blablabla.
- visszatérési érték lehet:
    - ModelAndView, Model, Map, View, String a View nevére, void
    - @ResponseBody-s handler esetén akármi
    - **HttpEntity<?>, HttpMessageConverterrel**
    - ResponseEntity<?>, Callable<?>, DeferredResult<?>

HttpMessageConverter:

- http kérés törzsét lehet leképezni vele objektummá
- `@RequestBody` és `@ResponseBody` annotációkhoz illék
- REST stílusú webalkalmazások esetén nagy a jelentössége

REpresentational State Transfer:
- olyan szoftver architektúra, amelyben különféle erőforrások URI alapon érhetők el
- kliens és szerver között olyan dokumentumok utaznak, melyek ezen erőforrások állapotain reprezentálják
- alapelv nem köti meg a reprezentáció formátumát, lehet XML, HTML, JSON, kép, szöveg, …

RESTful webszolgáltatás:
- HTTP fölött, URI alapon egyszerűen címezhető szolgáltatás, melyhez meg kell adni
  - a szolgáltatás URI-ját
  - a szolgáltatás által támogatott MIME typeot
  - a szolgáltatás által támogatott HTTP metódusokat (GET, POST, PUT, DELETE)
- nincs rá szabvány, mint pl XML webszolgáltatásoknál a SOAP meg a WSDL
- megfelelően konfigurált és megírt szervlettel is megvalósítható, de kényelmesebb keretrendszer támogatással, pl. Spring MVC
- Java EE 6 a Spring MVC-hez igen hasonló API-t standardizált: **JAX-RS**

model attribútumok élettartama: FlashMap
- milyen scope-ba tegyük az attribútumokat, ha redirectes navigáció is van?
  - request → nem marad meg
  - session → túl sokáig van ott, ha nem szedjük ki explicit módon
- **flash élettartamú attribútum**: csak a következő kérésig marad a modellben
- megvalósítása: minden requesthez tartozik két **FlashMap:**
  - **input:** előző request által eltárolt adatok
  - **output:** amit mi akarunk a következő request számára beletenni
- statikus metódussal bárhonnan elérhetők
- tipikus használatuk a kontrollerben még egyszerűbb:
  - input: automatikusan látszódnak a modellben
  - output: handler metódus `RedirectAttributes` bemenő paraméterén keresztül

validáció:
- **programozott validáció**: saját Validator interfész
- **deklaratív**: Java EE 6 által bevezetett Bean Validation használata

paraméterek konvertálása:
- **`@InitBinder`** annotációval regisztrálhatunk saját konvertert
- használata típus alapján automatikus
- controllerek közös kezelése: gyakori igény, hogy az összes controllerre egy helyen akarunk `@InitBinder` vagy `@ExceptionHandler` vagy `@ModelAttribute` metódust definiálni
  - erre megoldás a Spring MVC 3.2 óta a `@ControllerAdvice`: összesre érvényes lesz

view:
- JSP mellett támogatott a Velocity, FreeMarker, Thymeleaf, de integrálható Struts Tiles-al is
- view-k leképezése név alapján történik: `ViewResolver`-ek végzik a feladatot
- **témákkal** is lehet szórakozni, ezek kiválasztását egy `ThemeResolver` implementáció kezeli, ebből három beépített van:
  - `FixedThemeResolver` – egy témát állíthatunk be vele globálisan az összes usernek
  - `SessionThemeResolver` – user sessionjében tárolja a kiválasztott témát
  - `CookieThemeResolver` – usernél kliens oldalon tárolja a kiválasztott témát
- `themeResolver` néven kell konfigolni, és lehet `@Autowired`-el injektálni és `setThemeName` metódussal változtatni pl. userről tárolt adatok vagy user input alapján!

Spring Security

- authentikációt és authorizációt támogató keretrendszer
- rugalmasabb, mint a Java EE szabványban benne levő JAAS → a Spring Security konfigolható úgy, hogy JAAS-t használjon
- Springet használ, de bármilyen alkalmazásban felhasználható
- őse az Acegi Security System

authentikáció:

- JAAS-hoz hasonló koncepció: ha védett url-hez akar hozzáférni a user és még nincs authentikálva, bejelentkezteti a Spring Security
- **authentikációs adatok bekérése:**
  - saját form
  - böngésző által feldobott ablak: HTTP BASIC, HTTP DIGEST
- **authentikációs adatok tárolása:**
  - relációs adatbázisban (JDBC) → jelszó akár hashelve
    - valós környezetben ált. így tárolunk felhasználói adatokat, nem konfig fájlokban
  - LDAP
  - JAAS
  - OpenID
  - OAuth
- authentikációs felület testreszabása:
  - a HTTP BASIC authentikáció base64 kódolva küldi a jelszót, tehát a hálózatot lehallgatva visszafejthető (mondjuk a https azért kivédi)
  - a HTTP DIGEST authentikáció is beállítható, ekkor challenge-response alapú az authentikáció, maga a jelszó nem utazik a hálózaton (nehézkes bekonfigolni)
- egyéb felhasználói adatokat be tudunk tölteni authentikáció során, ha csinálunk egy `UserDetailsService` implementációt és felülcsapjuk a `loadUserByUsername` függvényét
- milyen extrákat tudunk még csinálni:
  - jelszó hash-elt tárolása
  - salting
  - remember-me
  - HTTPS kikényszerítése bizonyos url-mintákhoz, szerephez kötötten
  - lejárt session-ök kezelése általunk definiált url-el
  - egy user konkurens sessionjeinek maximalizálása

authorizáció:

- URL alapú:
  - **`http.authorizeRequests()`** → `antMatchers()`-el URL mintákra szabályok adhatók,
    - pl. permitAll, authenticated, hasRole, hasAnyRole, denyAll, rememberMe, hasIpAddress, hasAuthority, hasAnyAuthority, …
  - alapelv: felülről lefelé értékelődnek ki, ha egy szabály illeszkedik, nem megy tovább
  - URL alapú trükközésekre oda kell figyelni, mint pl. ../
- nem csak URL, hanem metódus szinten is deklarálhatunk megkötéseket:
  - `@Secured(„ROLE_ADMIN”)`
  - `@PreAuthorize(„hasAuthority('ROLE_TELLER')”)`

authorizáció és üzleti logika izolált tesztelése:

- ha csak az üzleti logikát akarjuk tesztelni, konfigban kikapcsoljuk a biztonsági ellenőrzéseket
- ha csak a biztonsági beállításokat akarjuk tesztelni, üres üzleti logikával, az is megoldható :3

szerepek és jogok:

- ha szerepeket ellenőrzünk a kódban, és bejön egy új, akkor sok helyen módosítgatni kell
- megoldás: kódban csak jogokat vizsgáljunk
- a jogok egy halmazát tekintsük role-nak, a userek role-okat kapjanak
- a Spring Security számára a jog és szerep ekvivalens, csak egy suttyó string
  - a role egy speciális authority, ami ROLE_-al kezdődik
  - Spring Security 4-től a ROLE_ prefixet automatikusan hozzáteszi, Spring Security 3-ban még explicit ki kellett tenni
- a default JDBC auth.privder DB szinten a groups táblával támogatja

domain instance based authorization:

- bizonyos metódusok meghívhatósága attól függ, milyen paraméterekkel hívjuk
  - pl. a user csak a saját foglalását tudja törölni → erre való a `@PreAuthorize`
- másik megoldás, hogy az objektumokkal együtt tároljuk, kik férhetnek hozzá (nehézkes)
- egy szebb megoldás:
  `@PreAuthorize(„hasAuthority('RIGHT_CANCEL') and` **`hasPermission`**`(#hr,'cancel')”)`
- lehet, hogy nem bemenő paraméter alapján akarunk ellenőrizni, hanem a visszatérési értéket akarjuk szűrni: **`@PostFilter`**`(„`**`hasPermission`**`(filterObject, 'list')”)`
  permission nélkül: `@PostFilter(„filterObject.username == authentication.name”)`

Spring Security JSP oldalakon:

- Spring Security taglib: `<sec:authentication property=”principal.username” />`
- jogosultságellenőrzés JSP nézetben:
  - cél: ki se tegyünk egy linket, ha a usernek nincs joga odanavigálni
  - megoldás: JSTL if + flag az objektumon

Spring Web Flow:

- oldalfolyamok hatékony kezelése, Spring MVC, JSF és Portlet környezetben is
- oldalfolyam = több oldalon át ívelő, varázsló-jellegű munkafolyamat
- kihívások egy állapottal rendelkező oldalfolyam megvalósításakor:
  - nehéz vizualizáció
  - `HttpSession` intenzív használata
  - flow-nak megfelelő navigáció kikényszerítése
  - böngésző back gomb kezelése
  - több böngésző tab által okozott konkurenciaproblémák
- a flow belépési és kilépési ponttal rendelkezik, lépésekből áll

action-ök: ha valamilyen ponton valamilyen üzleti logikát szeretnénk meghívni:

  - flow kezdete/vége
  - belépés állapotba / kilépés állapotból
  - nézet renderelése
  - állapotátmenetkor
- az action Unified Expression Language-ben / Spring EL-ben, OGNL-ben adható meg

nézetek renderelése:

- a nézetek default a flow-t definiáló xml mellett vannak
- de megadhatunk abszolút útvonalat is
- ha olyan frameworkkel (pl. Spring MVC, JSF) együtt használjuk, ami a view nevet konfig alapján feloldja, akkor az lép működésbe
- a nézet rendereltethető felugró ablakban (popup=”true”)
- szövegek properties fájlokkal lokalizálhatók
- változók tehetők viewScope-ba, addig él, míg át nem megyünk más view-ra (AJAX haver)

Thymeleaf

általános jellemzők:

- java library, XML/HTML/HTML5 template fájlok kezelésére
- rugalmasan bővíthető dialektusokkal, pl. Standard Dialect, SpringStandard Dialect
- a thymeleaf template szabványos XML/HTML, csak th: névtérbe tett **attribútumokkal** bővítve → statikus fájl template egyszerűen designolható
  - a designolás elvégezhető a template fájlon, nem kell szervert futtatni, appot telepíteni
  - a designer és fejlesztő ugyanazon a fájlon dolgozhat
- belső működés:
  - saját DOM implementáció
  - parszolt template-eket cacheli

party Spring Boot-tal:

- thymeleaf-spring4 classpathba
- HTML fájlokat classpathba, templates folder alá
- application.properties-be: spring.thymeleaf-cache=false

milyen szexi dolgokat tud még ez a thymeleaf:

- **${kif}** – változó kifejezések – Spring EL kifejezés, leggyakrabban modell objektumok
- *{} – selection kifejezések – egy gyökér objektumhoz relatív propertyk elérése
- #{} – I18N kifejezések properties fájlban lévő kulcsot adhatunk meg
- **th:text="#{welcome(${person.name})}"** – szövegek
  - escapelés: minden kifejezés HTML escapelődik default (ez ellen: **th:utext**)
- **th:each**"prod : ${prods}" – iteráció
- **th:block** – több elem generálása egy iterációban
- **th:if** – feltételes kiértékelés, negálható !-el és not-al is – van még **th:unless** is
- **th:switch**, **th:case** – switch-case

globális hibák:

- olyan hibák, amik nem konkrét input mezőhöz vannak rendelve

fragmentek kezelése:

- cél: oldalak felépítése kisebb darabokból, modularizálás
- <div **th:fragment**="copy">, aztán van ilyen is, hogy <div **th:include**="footer :: copy">
- **fontos:** a fragment minden modell változót használhat, nem csak azokat, amiket ő definiált paraméterként
- a paraméterek validálhatók a **th:assert** attribútummal

layout dialect:

- ha több oldal szeretné behúzni ugyanazt az elemet, redundancia jelentkezik:
  - lesz egy rakás th:include, meg egy rakás HTML váz, ami tartalmazza ezeket
- erre megoldás a layout dialect:
  - JSF facelets, Tapestry layout komponensekhez hasonló
  - segítségével egy layout oldal definiálható, ez:
    - definiálja a teljes HTML vázat
    - definiálja azokat a részeket, amiket az oldalak majd felüldefiniálhatnak
  - több layout oldalunk lehet, minden oldal kiválaszthatja, melyik layoutot követi

inlining:

- nem kötelező a szövegeket th:text attribútumba tenni, lehet inline is, akkor így néz ki:
  - <p> csá **[[${bela.neveneki}]]**! </p>
- Thymeleaf 3 előtt ezt engedélyezni kell valamelyik szülő tagben: **th:inline="text"**
- hátránya:, hogy statikus nézetben nem tudunk más, értelmes szöveget mutatni helyette

kommentek:

- a HTML komment statikusan és végrehajtás után is benne marad az oldal kódjában
- `<!--/* ez törlődik végrehajtáskor */-->`
- `<!--/*/ ez bent marad végrehajtás után /*/-->`
  - statikus nézetben forrásban ott van, de a böngésző nem jeleníti meg

Wicket

kihívások:

- UI és alkalmazás illesztése
  - konverzió (string → Java típus)
  - validáció (hossz, érték, egyezőség)
  - navigáció és URL kezelés (redirect, egy vagy több url, szép url-ek, …)
- újrafelhasználhatóság
  - komponens szinten (→ JSF component)
  - panel szinten (→ JSF composite component)
  - oldal szinten
- állapottárolás (állapotmentesség, állapottárolás)
- AJAX támogatás (komponens-integrált, egyedi)

válaszok a kihívásra:

- templatelő – minimalista
- komponens orientált (JSF, Apache Wicket)
- szeparáló (JS kliens + szerveren REST/WS interfész)
- egyesítő (Node.js, Scala.js)

wicket bevezetése:

- open source keretrendszer
- komponens orientált
  - újrahasznosítható osztályok
  - könnyebb fejlesztés, nehezebb debug
- modell alapú
- állapottároló (kézi / komponensek általi)

model interfész: MVC pattern – nem közvetlenül az entitásokat használjuk

- **modell mint tároló (container):**
  - IModel<T> interfész – getter, setter
  - maga az adat kerülhet a modell implementáción kívülre vagy belülre
  - standard implementáció vagy saját
  - PropertyModel, CompoundPropertyModel
- **modell mint adapter:**
  - összetett modelleknél
    - nincs 1-1 megfeleltethetőség a model java osztály és a komponensek között
  - egyszerű modelleknél
    - típuskonverzióra alternatíva az IConverter
- **modell mint adatbázis proxy:**
  - csak ID-t tárol
  - igény szerint betölt/elfelejt
- **modell mint erőforrás proxy:**
  - hasonló az adatbázis proxyhoz, csak nem adatbázist használ, hanem fájlt, streamet, properties fájlt (→ lokalizáció)

komponens hierarchia:

- fa struktúra a kódban:
  - gyökérben `Page*`
  - csomópontokban `MarkupContainer*`
  - levelekben `Component*`
- fa struktúra a markupban: a hierarchia meg kell egyezzen, de a gyerek csomópontok sorrendje tetszőleges
- **component:**
  - minden komponens őse
  - általános funkciók: egyedi id, visibility, enabled, … idk
  - életciklus callbackek: onBeforeRender, onComponentTag, onRemove
  - utility metódusok: getPage, getSession, setResponsePage, info, error, warning, ilyesmik
- **markupContainer:**
  - Map a gyerek komponenseknek
  - add(component), remove, removeAll
  - visitChildren → beépített visitor pattern
  - a legtöbb, önálló markup fájllal nem rendelkező komponens őse
- **page:**
  - speciális komponens, mely egy oldalt reprezentál
  - MarkupContainer → vannak gyerekei
  - tartozik hozzá egy markup fájl
  - tárolás session-ben
  - back-button támogatás
- **webMarkupContainerWithAssociatedMarkup:**
  - tartozik hozzá egy markup fájl

form komponensek:

- FormComponent gyerekosztályok
- kérés paramétereit dolgozza fel
- validálható
- HTML imput, select, textarea, stb. leképezései
- itt a modellnek már input-output szerepe van
- szövegbevitel: textfield (felismeri a típusokat), textarea
- select: dropdownchoice, listmultiplechoice, radiochoice

konverzió:

- IConverter interfész
- beépített konverterek Integer, Long, Float, Boolean, … típusokra
- lehet saját konvertert is implementálni

validáció:

- IValidator<T> interfész – validate() metódus
- beépített validátorok: Minimum/Maximum/LengthBetween/Pattern/DateValidator, …
- de itt is írható saját :3

üzenetkezelés:

- üzenet jön létre,
  - ha hiba történik a keretrendszerben, vagy a saját kódunkban
  - csak informálunk
- tárolás sessionben
- megjelenítés: FeedbackPanel, ComponentFeedbackPanel, IHateLife

lokalizáció:

- properties fájlokkal – hierarchikus keresés

wicketFilter:

- framework belépési pontja
- web.xml-ben kell beállítai

application osztály:

- egész alkalmazásra közös konfigurációs osztály
- init() metódus
- rakás beállítás, pl. getPageSettings, getFrameworkSettings, getExceptionSettings, …
- newSession metódus → session létrehozása, ó erre nem gondoltam volna :o

wicket session:

- típusos session osztály
- **PRO:** wicket komponensekből könnyen elérhető
- **CON:** csak a keretrendszer hozhatja létre

kérésfeldolgozás:

1. kell-e kérésfeldolgozás?
2. kérés előkészítése
3. kérés céljának megállapítása
4. események kezelése
5. válasz generálása

AJAX:

- nem ajaxosan működő komponensek újrahasznosítása: textField, DropDownChoice, stb..
  - Fallback
- egy oldal életciklusa:
  1. nem Page(...)
  2. AJAX-os hívások
  3. elnavigálás
  4. visszatérés (back button vagy link)

JavaServer Faces

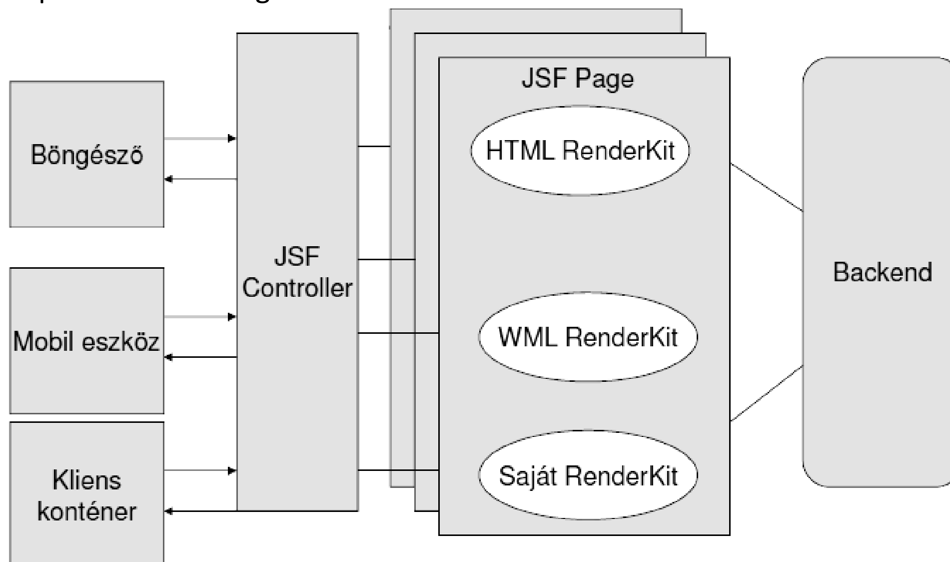
JSF háttér:

- webappok jelentik a leggyakoribb belépési pontot a Java EE területre
- a Java EE egyszerűbb webes technológiával tulajdonképpen bármi megoldható, de nem a legkényelmesebb sajnos :(
- egy kezdő fejlesztő nem biztos, hogy meg akarja/tudja keresni a számára legkedvezőbb webes keretrendszert, amivel egyszerűsíti a fejlesztést →
- a számos, addigra már meglévő keretrendszer tapasztalataira építve a Java EE 5 definiált egy webalkalmazás keretrendszert = **JavaServer Faces**
- szabvány része → minden alkalmazásszerverben van implementáció

JavaServer Faces:

- egy szerver oldali, komponens alapú felhasználói felület keretrendszer, webes és általános környezetre:
  - egy létező igényt és űrt tölt ki
  - úgy tervezték meg, hogy egyszerű legyen hozzá eszköztámogatást biztosítani
  - szabvány, nagy ipari támogatással

- **miért jó ez:**
  - MVC, UI-koncepció webes környezetben, a korábbi tapasztalatokra építve
  - komponens alapú, támogatást biztosít a kliens megjelenítéstől független viselkedésre
  - finomabban hangolható, mint a JSP, a UI elemek állapottal rendelkeznek szerver oldalon
  - konkrét megjelenítési technológiától független, de úgy tervezték meg, hogy JSP környezetben és annak hiányában is tudjon működni
  - a felületi komponensek állapota, eseménymodellje, renderelése jól specifikált
  - Java EE része, minden alkalmazásszerver tartalmaz implementációt
  - számos ingyenes komponenskönyvtár érhető el hozzá, sok hasznos UI komponenssel
  - újrafelhasználható összetett komponensek fejlesztése egyszerű
  - beépített AJAX támogatás



alapvető JSF képességek:
- *UI komponens modell*: megjelenítés-független működési jellemzők
- *rugalmas rendering modell*: nem csak HTML rendering definiálható
- *eseménykezelés*: hagyományos ablakos alkalmazásoktól megszokott módon
- *validáció*: deklaratív és programozott egyaránt
- *deklaratív oldal-navigáció*
- *I18N*
- *accessibility*

JSF szerepkörök:
- *page author*: webes alkalmazás felhasználói felülete
- *component writer*: újrafelhasználható komponensek írása
- *application developer*: szerver oldali funkcionalitás UI-tól elkülönülő része
- *tool provider*: nem fejlesztő, hanem támogató eszközök, IDE a fentiekhez

JSF verziók:
- Java EE 6: JSF 1.2 – annotációk helyett kötelezően xml
- Java EE 6: JSF 2.0 (+2.1) – ebben sok újítás:
  - annotációk
  - facelets
  - AJAX támogatás
  - implicit navigáció
- Jave EE 7: JSF 2.2

JSF életciklus:

- JSF oldal = UI komponensekből álló fa (view)
- életciklus lépései:
  1. komponensfa felépítése (view)
  2. request értékek beállítása
  3. validáció
  4. modell frissítése
  5. alkalmazás-események megívása
  6. válasz renderelése
- életciklusnál két kéréstípust különböztetünk meg:
  - **request:** csak 1. és 6. lépés
  - **postback:** minden lépés végén végrehajtódik

1. komponensfa felépítése (view)

- objektum-hierarchia felépítése
  - első kérés esetén a nézet az oldalon elhelyezett custom tagek alapján épül fel
  - postback esetén a view-t és az állapotát viewstate-ből állítja vissza
    - ami általában a view objektum szerializált változata
    - és két módon tárolható:
      - **client:** teljes viewstate a kliens oldalon tárolódik, a HTML-ben hidden inputként, submitkor bisszamegy a szerverre
      - **server:** a viewstate szerver oldalon tárolódik a memóriában, csak egy viewId megy le a kliens oldalra, ez az default!
- eseménykezelők, validátorok „bedrótozása”
- view elmentése FacesContext-en belül
- nem-postback esetben ugrás a 6. fázishoz!

2. request értékek beállítása

- minden UIInput komponens (kivéve rendered=”false”) kinyeri a request-ből az új értékét
- ezt az értéket string-ként tárolja a komponens, és későbbi feldolgozás során el lehet érni
- az ActionSource típusú komponensek is kinyerik a requestből, hogy meg lettek-e nyomva, és ha igen, megfelelő ActionEventet helyeznek el az eseménysorban

3. validáció

- JSF implementáció minden regisztrált bemeneti validátort meghív a komponensfán
- konverzió és validáció történik
- hibaüzenetek a FacesContext-be kerülnek
- hiba esetén rögtön a 6. pontra ugrik a feldolgozás

4. modell frissítése

- JSF implementáció végigmegy a komponensfán és a komponensek lokális validált értékeit beírja a value attribútummal a komponenshez kötött alkalmazásspecifikus objektumba
- típuskonverzió, illetve konverziós hiba történhez iss, ilyenkor hibaüzenet kerül a FacesContextbe, és megint ugrik a render fázisra

5. alkalmazás-események megívása

- előző fázisok során felhamozódott események továbbítása a megfelelő eseménykezelőknek
- ezek jellemzően gombok, linkek eseménykezelő metódusai és a UIInputhoz-hoz rendelt valueChanged eseménykezelők
- ezek megírásakor már a modellben lévő validált értékekkel dolgozhatunk

6. válasz renderelése

- az implementáció a komponensek beépített encode metódusán keresztül felépíti a komponens-fa megfelelő leírását
- ha itt hiba történik, akkor még elő tudja szedni a korábbi renderelt állapotot, és azt egészíti ki a jelenlegi hibakóddal! :o
- itt minden esetben meghívásra kerül a megfelelő RenderKit
- elmenti az állapotot és a következő kérések feldolgozásakor el tudja érni az 1. fázisban

alapértelmezett életciklus módosítása:

- ezt a remek életciklust a FacesServlet biztosítja = a kérésnek a web.xml-ben beállított URL mintára kell érkeznie
- lehetőségünk van arra is, hogy **JSF-es kérés nem JSF-es választ generáljon:**
  - JSF-es appban mindig elérhető a servlet/portlet API
  - ezen keresztül is összeállíthatjuk a választ
  - jelezni kell a `FacesContext.responseComplete()` hívással, hogy a 6. fázis ne rendereljen
- vagy hogy **nem JSF-es kérés JSF-es választ generáljon:**
  - viszonylag ritkán használt, tulajdonképpen a FacesServlet tevékenységét kell nekünk végrehajtani, de van rá API
  - JSF-es objektumokra kell referenciát szerezni
  - új view-t kell létrehozni, felépíteni
- JSF-es kérésre JSF-es válasz esetén is tudunk módosítani ezt-azt

komponensek renderelése:

- nem a komponens renderel, hanem a RenderKit
  - a komponens viselkedését csak egyszer definiáljuk
  - a komponenshez több megjelenítés is tartozhat
- a JSF oldal szerkesztői a komponenshez tartozó megfelelő tag kiválasztásával befolyásolhatják a megjelenítést
- **RenderKit**: a komponens osztályok és a komponenst leíró tag-ek kapcsolatát írja le
  - a JSF implementációk beépített RenderKit-et tartalmaznak HTML kliensekhez
  - minden pomponenshez **Renderer objektumokat** társít:
    - valamilyen módszert definiál a komponens megjelenítésére adott RenderKiten belül

JSF tag libraryk:

- `html_basic`: alapvető HTML komponensek, HTML környezetre illesztve
- `jsf_core`: általános komponensek, megjelenítő környezettől függetlenül
- komponensek: mindegyiknek van egy id-je, ha nem adjuk meg attribútumként, generálódik

JSF 2.0 – Facelets:

- JSP-JSF együttműködéssel voltak bajok
- a JSF kezdettől fogva jól pluginolható, nem csak új RenderKitek, hanem új ViewHandlerek is írhatók, amikkel az oldalak tetszőleges saját nyelven is összeállíthatók
- elterjedt technológia a JSF oldalak definíciójára = **Facelets**, ennek főbb előnyei:
  - XHTML → bármilyen HTML editorral szerkeszthető
  - absztrakt szintaxisfává fordul, amely végrehajtáskor közvetlenül UIComponent-ek hierarchiáját állítja elő (nem fordul servletté, mint a JSP, hatékonyabb)
  - jobb hibajelentések fejlesztési időben
  - templatek segítségével egyszerűbb az oldalak felépítése kisebb egységekből, jobb újrafelhasználás
  - EL validáció fordítási időben
  - saját tag definiálása egyszerűen megoldható, TLD nélkül

- JAF 2.0 előírja a Facelets támogatását, nem kell külön libraryként hozzáadni
- JSP-t deprecatednek minősíti JSF használatakor, minden új JSF feature csak Facelets-el megy

Facelets tagek:
- **`f:`**, **`h:`** JSF komponensek definiálására
- **`c:`** JSTL core tagek egy része (`c:if`, `when`, `otherwise`, `choose`, `forEach`, `set`, `catch`)
- **`fn:`** JSTL függvények
- **`ui:`** templatekezelés (`ui:insert`, `define`, `composition`, `include`, `decorate`, `param`)
    - **`ui:composition`** ↔ **`ui:decorate`**:
        - mindkettő hivatkozhat egy template-re, aminek `ui:insert` elemeit `ui:define` elemekkel töltik ki
        - `composition` esetén a rajta kívül levő elemek nem fognak renderelődni
        - `decorate`-nél fognak
        - ugyanez a különbség `fragment` és `component` között is (`component` levág)
    - komponensdefiniálás (`ui:component`, `fragment`)
    - debugolás (`ui:debug`)
    - iteráció (`ui:repeat`)
    - view egy részének eltávolítása már fordítási időben (`ui:remove`)

how to comment:
- JSP komment nem működik
- XML komment nem a várt módon működik (kirendereli a belsejét is!! D:< outrage)
- szóval ha komponenst akarsz kommentezni, tedd bele `<ui:remove>`-ba szt jó van

extra jellegzetességek Facelets-el:
- használhatók Facelets oldalakon a JSTL tagek, de csak **core** és **függvények** (`c:`, `fn:`)
- `fmt` tagek helyett JSF `<f:loadBundle>` és `<f:formatXXX>` tagek vannak
- xml és sql nem is a nézet feladatai közé tartoznak!
- **fontos**: JSTL tagek csak akkor értékelődnek ki, amikor a view először felépül
  (1. fázis, nem postback)
  a többi tag általában komponenst helyez el a fában, és azok ott élnek mind a 6 fázisban :(

managed bean:
- szerver oldali, a felhasználói felülettel összekapcsolt POJO (≈ MVC model)
- faces-config.xml-ben, vagy annotációval adható meg
- állapotot reprezentáló propertyket vagy komponenspéldányt tárolhatunk benne
- a komponenshez tartozó eseménykezelő, validáló, ill. az oldal-navigációt vezérlő metódusok is elhelyezhetők benne
- JSF oldalon Expression Language kifejezéssel hivatkozhatunk propertyjére vagy metódusára
- managed beant első hivatkozáskor a JSF implementáció
    - példányosítja argumentum nélküli ctorral
    - beállítja rajta a menedzselt propertyket
    - elvégzi az `@EJB` és `@Resource` injektálásokat
    - majd meghívja a `@PostConstruct`-tal jelölt metódust, ha van ilyen
    - végül a megfelelő scope-ban tárolja, és legközelebbi hivatkozásnál onnan szedi elő
- a managed bean mindig valamilyen scope-ban él (request, session, application, custom, ...)
    - **none scope:** minden hivatkozásnál létrejön → ha injektálunk, élettartama megegyezik a tulajdonos beanével
    - **view scope:** request és session közötti élettartam, amíg el nem navigálunk az oldalról (AJAX-os kéréseknél hasznos)
    - **flash scope:** request scope-nál egy redirect-nyivel hosszabb ideig él

hogy lehet egy managed beant elérni:

- Facelets oldalon EL-ből: `#{ meBean }`
- más JSF managed beanbe tagváltozóként injektálható, ha nem szűkebb a scope-ja
- bármilyen osztályból programozottan, ha faces kontextusban vagyunk

navigációs lehetőségek:

- **deklaratív** navigáció – XML alapú
  - JSF 2.0 előtt kötelező volt
  - `WEB-INF/faces-config.xml`-ben
  - navigációs szabályok azt tartalmazzák, hogy melyik oldal következzen, ha egy action kimenete egy bizonyos string
  - tulajdonképpen string-oldal párosok
  - kötődhetnek egy konkrét oldalhoz, de lehetnek minden oldalra is érvényesek
  - minden navigáció egyszerűen átkonfigolható redirect-esre
  - az eseménykezelő string visszatérési értéke szabályozza a navigációt
- **implicit** navigáció – kényelmes, egyszerű
  - **JSF 2.0** újdonság
  - ha a JSF nem talál a `faces-config.xml`-ben megfelelő navigációs szabályt, az outcome stringet közvetlenül a cél oldalként értelmezi →
  - gyorsabb fejlesztés, de navigációs logika szorosan csatolt (xml-ben még átkofigolható)
  - ez is lehet redirectes
- **feltételes** navigáció – XML alapú, kerülendő a túlzott használata
  - ez is **JSF 2.0** óta új
  - XML-ben túlburjánzó feltételek helyett a navigációs logika kódba is helyezhető
- **preemptív** navigáció – GET típusú kérések, bookmarkolható linkek támogatása
  - bookmarkolható linkek generálása
  - nem JSF-es kérésre irányít át, de kihasználja a deklaratív navigációt
  - a navigációs logika már a link renderelésekor kiértékelődik
  - GET támogatás bulika:
    - JSF korábban csak a POST-ot támogatta
    - ennek fő hátránya, hogy nem voltak bookmarkolható utl-ek
    - **JSF 2.0** óta van `f:viewParam` tag, amivel le lehet képezni a request paramétereket managed beanek propertyjeire

eseménykezelés:

- **JSF 1-es események:**
  - alkalmazás szintű események: actionEvent, valueChangedEvent, phaseEvent (életciklus)
- **JSF 2: SystemEventek:**
  - olyasmi, mint a phaseEvent, nem alkalmazásspecifikus, de finomabb granularitású (Pre/PostConstructApplicationEvent, …)
  - listenert lehet regisztrálni programozottan vagy deklaratívan

konverterek:

- vannak a JSF-ben beépített konverterek: BigDecimal, BigInteger, Boolean, Char, DateTime,...
- ezek közül néhány automatikusan működésbe lép, ha megfelelő típusú managed bean propertyhez kötjük
- dátumokra viszont explicit ki kell rakni az oldalon patternnel (néha számra is van értelme
- írhatunk sajátot is: implementálni kell a `javax.faces.convert.Converter` interfészt!
  - két csuda metódusát kell felülcsapni: `getAsObject()` és `getAsString()`
  - a konvertert a faces-configban lehet regisztrálni, vagy annotációval

validátorok: négy módon végezhető validáció:

- beépített validátorokkal (deklaratívan)
  - nincs túl sok, Range, Length, ilyesmik.
  - A JSF komponenskönyvtárak általában tartalmazzák az összetettebbeket (pl. regex)
  - használhatók az oldalon deklaratívan tagekben
- alkalmazásszintű validációval
  - az eseménykezelő metódusban, programozottan validálunk
  - komplexebb, üzleti logika jellegű validációt érdemes így elvégezni (pl. létező)
  - az eseménykezelő csak a többi típusú validáció után fut le, ha már korábban validálási hiba keletkezett, akkor nem jut el idáig :3
- inline validáló metódussal
  - managed bean-ben egy void függvény
  - `<h:inputText validator="#{mahbean.validateSomthing}" />`
- saját validátorral
  - a `Validator` interfészt kell megvalósítani egy `void validate()` metódussal
  - az elkészült validátort regisztrálni kell pl. annotációval
  - aztán az oldalon be lehet csapni egy `f:validator` taggel

komponenskönyvtár példa:

- saját komponens írása nehézkes, a JSF 2.0 egyszerűsít rajta
- de minek írnánk sajátot, ha van már csomó kész komponenskönyvtár, mint pl. a **MyFaces**?
  - ezek többsége open source, ingyé használható

JSF 2.0 – összetett komponensek:

- na hát szóval mint az az előbb elhangzott, JSF kezdetén is lehetett saját komponenst írni, de elég nehéz volt:
  - UIComponentből származtatott osztály
  - komponens és megvalósító osztály összerendelése faces-config.xml-ben
  - Renderer megírása
  - Renderer deklarálása és implementációhoz rendelése a faces-config.xml-ben
  - JSP vagy Faselets tag handlet megírása
- **good guy JSF 2.0** összetett komponensei jóval egyszerűbbek, mert Java kód és XML konfigurálás nélkül fejleszthetőek!
  - A komponenseket a resources könyvtárba kell menteni (xhtml)

AJAX támogatás:

- AJAX csak magában: kliens oldali felület interaktívabbá tétele
- **JSF és AJAX:**
  - JSF komponensek egyszerűen kiegészíthetőek JS megjelenítési elemekkel és a szerver oldalon is egyszerű az XML feldolgozás
  - Java EE környezetben a JSF UI koncepciója jól illeszthető az AJAX-hoz
  - a JSF komponensek egyszerűen kiegészíthetőek JS megjelenítési elemekkel, így AJAX-os JSF komponenseket is fejleszthetünk
  - JSF-es custom tagek teljesen elrejthetik a JS kódot
  - AJAX-os JSF komponenseknél általában megadható, melyik része küldődjön el a kérésben, és az oldal melyik része frissüljön, az alacsony szintű részletektől mentesen
  - a JSF csak a 2.0-tól standardizálja az AJAX-ot, de a komponenskönyvtárak már a korábbi verziókhoz is tartalmaznak AJAX támogatást :3

- **JSF 2.0 és AJAX:**
  - az eddigi AJAX támogatott JSF komponenskönyvtárak mind a saját koncepciójukat követték, nem igazán lehetett őket vegyesen használni
  - a JSF 2 standardizál egy JS könyvtárat és definiál egy AJAX életciklust részleges nézetbejárással és oldalfrissítéssel
  - nem írja elő, milyen stratégiával kell egy implementációnak részlegesen bejárnia vagy renderelnie a komponenseket (de azt igen, hogy milyen API-n keresztül)
  - AJAX-os működés elérhető a standardizált JS API-val, vagy deklaratívan (**`f:ajax`** tag)
  - a JS API annyival tud többet egy általános JS librarynél, hogy ismeri a JSF életciklust, vagyis kommunikálni tud a szerver oldali komponensfa elemeivel, így az szinkronban marad a kliens oldali DOM-mal
  - a `javax.jsf.request` a legfontosabb függvény, ebben megadható, mely komponensekre fusson le a JSF életciklus eleje (`execute`) és vége (`render`)

JSF 2.0 – kliens viselkedések:

- az `f:ajax`, validátorok és konverterek közös jellemzője: tetszőleges komponenshez hozzáadhatók, és így bővíti a szülő komponens viselkedését
  - a konverter és validátor kifejezetten szerver oldali funkcionalitást csatol
  - az `f:ajax` kliens oldali eseményre iratkozik fel, aminek hatására valamilyen JS fut le
  - ez a koncepció általánosítható a felelősségek szétválasztásával:
    - `ClientBehavior` feladata a komponensfüggetlen JS kód előállítása
    - a komponens feladata a JS elkérése a `ClientBehavior`-től, és elhelyezése komponens HTML-jében
  - az `AjaxBehavior` is tulajdonképpen egy `ClientBehavior` :O
- lehetséges felhasználások:
  - kliens oldali validáció
  - DOM manipuláció
  - animációk, effektek
  - confirm, alert ablakok
  - billentyűkezelés
  - adatok lusta betöltése
  - kliens oldali logolás

JSF 2.0 – erőforrások:

- a komponensek nagy része használ valamilyen erőforrást
- az oldalnak, ahol egy komponens szerepel, tartalmaznia kell hivatkozást ezekre a fájlokra
- ezek kiszolgálására korábban külön szervletet/filtert használtak a komponenskönyvtárak
- most 2 szabványos helyről töltődnek be – ebben a sorrendben:
  - a webapp resources könyvtárából
  - vagy a classpath-ban lévő jar-ok META-INF/resources könyvtáraiból
- a locale és verziófeloldást a JSF implementáció végzi

unified EL lehetőségek:

- EL-ben metódus átadása custom tag attribútum ként
- EL szintaxis bővítése saját EL Resolver írásával
- néhány standard resolver készen: tömbök, JavaBeanek, listák, map-ek értelmezésére
- saját resolver ezek után regisztrálható be

Java EE 6 – EL 2.2:

- nem JSF-specifikus, JSP-ben is használható
- metódusok kaphatnak bemenő paramétert

JSF 2.2 áttekintéss:

- **JSR 344**
- **HTML5 támogatás**
    - eddig a JSF kifejezetten el akarta rejteni a HTML részleteit
    - mostantól minél inkább ki akarjuk használni a HTML-t
    - standard HTML tageket használhatunk, amiket a JSF csak akkor fog kezelni, ha JSF névtérben lévő attribútumot is hozzáadunk
    - megadhatók olyan attribútumok, amiket a HTML5-ös böngészők értelmezni tudnak, a JSF érintetlenül hagyja őket („pass through" attribútumok és elemek)
- **erőforrás könyvtár szerződések**
    - a facelets template mechanizmusának továbbfejlesztése
        - template fájl definiálja a pontokat, ahol tartalmat lehet beszúrni (`ui:define`)
        - template-re hivatkozó fájl ezeket a helyeket tölti ki (`ui:insert`)
    - contract = templatek + beszúrási pontok + erőforrások
    - innen: /contracts könyvtár, vagy WEB-INF/lib jar-jai
- **Faces Flow**
    - cél: összetartozó oldalak összekapcsolása (pl. varázslók, workflow-k)
    - előzmények:
        - ADF Task Flows
        - Spring Web Flow
        - Apache MyFaces CODI
    - kb. olyan, mint egy metódus:
        - az app bármelyik pontjáról meghívhatók
        - flow-ból másik flow is meghívható és oda fog visszatérni
        - egy belépési pontjuk van
        - bemenő paramétereik és visszatérési értékeik vannak, szerződésben leírva
        - saját scope-baj tárolhatnak beaneket
    - navigáció most már nemcsak oldalak között mehet
    - csomópont típusok: View, Method Call, Switch, Flow Call, Flow Return
    - Flow-k definiálása: XML-ben vagy programozotan (Builder API)
- **stateless views**
    - ha nem akarjuk, hogy a view-hoz a szerver állapotot tároljon, mert
        - jobb teljesítményt akarunk
        - klaszterezett környezetben bármelyik csomóponthoz tudjuk a kéréseket irányítani
        - viewExpiredException megelőzése
    - `transient="true"` attribútum

Google Web Toolkit

áttekintés:

- webalkalmazás keretrendszer, amit a Gogol fejleszt és használ is (2.5-től open source)
- fő jellegzetessége: Java nyelvű fejlesztés GWT API-ra építve, amiből JS-t fordít a compiler
- a kész apphoz nem kötelező Java webkonténer, a generált JS és a JS-t behúzó HTML akár egy statikus webszerveren is elhelyezhetők
- nagyobb appok esetén szükséges lehet szerver oldali adatok elérése, erre kétféle támogatás:
    - GWT RPC → Java szervlet kell a szerver oldalon
    - JSON vagy XML → szerver oldal akármi lehet

GWT főbb előnyei:

- Javában írható a kliens oldali kód
- böngésző pluginnel megoldható, hogy a kliens oldal debugolása is a Java kódban történjen
- más-más böngészőspecifikus JS fordítódik a legelterjedtebb böngészőkhöz
- megvan a lehetőség saját JS kód írására is
- az SDK tartalmaz eszközt a JS teljesítményének analizálására
- AJAX támogatás magasabb absztrakciós szinten

GWT app felépítése:

- Host HTML:
  - ez tartalmazza az alkalmazást, a felhasználóknak ezt kell böngészőben megnyitni
  - behúzza a szükséges CSS, JS fájlokat
  - a törzs lehet akár üres is, a GWT-s widgetekből álló GUI-t majd a JS hozza létre
  - gyakran tartalmaz egy rejtett IFRAMEt, ami a history kezeléséhez szükséges
  - a törzsbe tehetünk tetsz. HTML tageket, amikbe akár GWT-s widgeteket is tölthetünk
- Modul XML:
  - GWT-s modulok konfigurációs egységek, saját modul XML-el
  - ennek kiterjesztése .gwt.xml
  - a classpath package struktúrájába illeszkedik
  - package-kvalifikált, kiterjesztés nélküli neve a modul alapértelmezett logikai neve
  - a modul XML-t a GWT compiler használja, futási időben nem szükséges

GWT compiler:

- a GWT SDK része, futtatható parancssorból, Eclipse pluginből, ant targetként
- a modul XML által hivatkozott Java osztályokat próbálja JS-é fordítani, ez csak bizonyos megkötésekkel sikerül
- a modul nevével megegyező könyvtárba generálja a kimenetet
- **bootstrap folyamat:** jó, de mit tesz a könyvtárba?
  - **<md5>.cache.html**
    - a fájl nevében a GWT kódból számított MD5 hash szerepel
    - egy ilyen fájl egy böngészőverzióhoz tartozó JS kódot tartalmazza
    - minden támogatott böngészőhöz külön cache fájl generálódik
  - **modulneve.nocache.js**
    - ez a bootstrap fájl, a host HTML erre hivatkozik
    - letöltésekor rejtett iframet hoz létre
    - ebbe nem-blokkoló módon betölti a böngészőverziónak megfelelő cache fájlt
  - folyamat előnyei:
    - a klienshez csak a saját böngészőjéhez tartozó JS kód fog letöltődni
    - cache stratégia valósítható meg a fájlokra
      - .cache → nyugodtan cachelhető
      - .nocache → nem szabad!!!!
- compiler egyéb kimenetei:
  - **hosted.html:** development módban ez töltődik be a böngészőbe
  - **<md5>.gwt.rpc:** GWT RPC használata esetén ebben van szerializálós policy
  - **örökölt témák design elemei**

futtatási módok:

- **production**:
  - a generált JS fut a kliens oldali böngészőben
  - az eredeti Java kódhoz már semmi köze, nem is kötelező Java webkonténerre telepítei

- **development**:
  - indítható parancssorból, vagy eclipse pluginként
  - szerveren Java bájtkódként fut a kliens oldali kód, ami így Java kódként debugolható
  - a böngészőben telepíteni kell a GWT developer plugint, ami a szerver oldalon futó kliens-oldali kódnak megfelelően alakítja a GUI-t
  - url-ben át kell adni a kód szerver ulr-jét
  - régebben a hosted mód külön alkalmazást indított, most bármilyen böngészőben futhat, amihez írtak pluging → a tényleges célböngészőn lehet tesztelni
  - ha a tényleges szerver oldali kód is Java, az vagy a kód szerveren fut, vagy akár külön szerveren, egyértelműen Java kódként debugolható
- **super dev mode:**
  - böngésző plugin nélkül megy, a Java kódot a böngésző Developer Tools nézetében debugolhatjuk → működés alapja a Source Maps: a generált JS-ből visszaállítja azt, amiből keletkezett
  - **PRO:**
    - böngésző API hibák nem érintik
    - nem kell új plugin verzióra várni a böngésző frissítése után
    - együtt debugolható a JS kóddal
    - sok JS kód esetén sokkal gyorsabb
  - **CON:**
    - experimental, Chrome és Firefox támogatják csak
    - sok Java debugger feature nem érhető el
    - a szerver oldal nem futtatható benne, azt külön szerveren kell

kliens oldali Java kód sajátságai:

- GWT 1.5-től támogatott a Java SE 5 (annotációk, generikusok, ...)
- Java SE funkcionalitás nagy része nem értelmezhető böngészőben → a GWT a standard Java-nak csak egy részhalmazát támogatja a kliens kódban
- development módban jelzi a nem támogatott osztályok használatát
- a szerver oldalon futó kód természetesen komplett Java EE lehet
- **numerikus típusok**:
  - JS-ben csak egyféle szám van, egy suttyó 64 bites lebegőpontos → ez egészeken más és a float-on végzett műveletek is double precizitással hajtódnak végre
  - a long kivétel, azt két 32 bitesként emulálja → lassabb
- **kivételkezelés**:
  - try, catch, finally használható
  - getStackTrace nem támogatott
  - több kivétel JavaScriptException formájában jelenik csak meg
- **assertion**: dev módban aktívak, prod módban `-ea` compiler kapcsolóval aktiválhatók
- **szálkezelés**: JS egyszálú → wait, notify, notifyAll nem támogatott, synchronized megengedett, bár nincs hatása
- **reflection**: nem támogatott, csak a típus lekérdezése
- **finalizer**: nem támogatott
- **reguláris kifejezések**: JS regexként értelmeződnek (nem pont ugyanaz, mint a Java-s)
- **szerializáció**: standard Java szerializáció nem vitező át JS-re, helyette van GWT RPC valami
- **időzítés**: saját osztályok: Timer, DeferredCommant, IncrementalCommand

kliens-szerver kommunikáció:

- java webkonténert tartalmazó szerver oldal esetén a GWT RPC magas szintű támogatást nyújt aszinkron hívásokhoz
- más szerverek elérése JSON/XML formátumú adatokkal történhez

GWT RPC:

- HTTP fölött működő távoli eljáráshívás
- az adatformátum pehelysúlyúbb pl. a SOAP-hoz képest
- fejlesztőként megírandó:
  - távoli interfész
  - implementációs szervlet
  - aszinkron interfész, amelyet a kliens oldal lát majd
- szerializáció:
  - minden bemenő paraméternek és visszatérési értéknek szerializálhatónak kell lennie
  - a GWT szerializálni tud:
    - primitív típust / wrapper osztályát
    - szerializálható típusok tömbjét
    - egy osztály akkor szerializálható, ha
      - a `java.io.Serializable` vagy a googles `isSerializable` marker interfészt implementálja
      - ÉS a nem-final, nem-transient példányváltozói szerializálhatók
      - ÉS van akármilyen láthatóságú no-arg konstktora

JPA (+ JDO) entitások szerializációja:

- több perzisztancia provider bájtkódmódosítást hajt végre az entitás osztályokon
- az ilyen enhanced osztályoknál eltért az osztálydefiníció kliens és szerver oldalon
- ezeket 2.0-tól tudja szerializálni a GWT
- GWT automatikusan enhanced-nek tekinti a JPA és JDO entitásokat
- ezen felül a modul XML-ben lehet megjelölni enhanced osztályokat
- **enhanced osztályok szerializálása:**
  - a nem-enhanced tagváltozókra normál módon
  - a nem-statikus, nem-tranziens enhanced tagváltozók szerver oldalon egyetlen értékké Java-szerializálódnak, és lemennek kliens oldalra, amely érintetlenül hagyja
  - ha a kliens oldal visszaküldi, a GWT szétbontja és deszerializálja ezt az értéket, majd setterekkel, azok hiányában közvetlenül beállítja a megfelelő enhanced tagváltozókat
- **feltételek:**
  - a szerlializálandó példánynak lecsatolt állapotban kell lenni
  - nem-statikus, nem-tranziens enhanced tagváltozóknak Java-szerializálhatónak kell lenni
  - ha egy enhanced tagváltozó módosításának mellékhatása van, azt JavaBean setterben kell implementálni
- **szerializálás többes kapcsolatokkal:**
  - ha egy JPA entitás Collection típusú tagváltozót tartalmaz, az általában a perzisztencia provider saját implementációja
  - ezeket nem tudja szerializálni a GWT, de van pár megoldás rá:

- **Data Transfer Object:**
  - olyan osztály, ami az entitás attribútumait tartalmazza, de nem perzisztens
  - létrehozásakor egy entitás példány tagváltozóit másoljuk bele, de a Collection-öket egyszerű implementációkra cseréljük le
  - **CON:**
    - plusz kód a DTO osztályok és a létrehozásuk megvalósítására
    - az átmásolás overhead-je jelentős lehet nagy objektumgráfoknál
  - kis egyszerűsítés: ha csak üres collection-öket kell lecserélni, ahhoz nem kell külön DTO osztály, elég lehet null-ra állítani a tagváltozókat
- **Dozer:**
  - XML formátumú mappingfájlok alapján automatikusan generál DTO osztályokat
  - a DTO létrehozását, az adatok átmásolását is megvalósítja
  - a DTO fejlesztését kiküszöböli, de a futási idejű overhead megmarad
- **RequestFactory:**
  - GWT 2.1 beépített megoldása DTO objektumok generálására
  - entitásokhoz proxy interfészt kell írni
  - EntityProxy-k használatakor a GWT RPC helyett RequestFactory-specifikus kommunikáció fog lezajlani, ehhez tartozik:
    - egy service stubokat gyártó interfész,
    - minden service-hez egy stub interfész,
    - szerver oldali implementáció
  - lényeg: a kliens oldali kód mindig csak az EntityProxy-kat fogja használni, amivel nem lesz szerializálási probléma
- **Gilead:**
  - open source library, korábbi neve Hibernate4GWT
  - alapelv több JPA perzisztencia proivderre mevalósítható, de egyelőre csak Hibernate-hez van implementálva
  - működése:
    - konfigurációt igényel
    - lecseréli a collection-öket alap implementációkra
    - eltárolja az infót, ami alapján vissza tudja cserélni az eredeti perzisztens collection-ökre db-elérés nélkül
  - kódban is módosítást igényel:
    - LightEntity-ből kell származnia az entitásnak
    - PersistenceRemoteService-ből kell származnia a GWT szolgáltatásnak

JSON alapú kommunikáció:

- a szerver oldali kód akármi lehet, a lényeg, hogy egy megfelelően paraméterezett HTTP kérésre megfelelő JSON-t adjon vissza
- a kliens oldali Java kódban
  - elküldjük a HTTP kérést aszinkron módon
  - fogadjuk a választ
  - parszoljuk a JSON stringet

JSON parszolása → JavaScript Native Interface:

- célja: olyan Java metódust írni, melynek implementációja JS-ben van
- egy JSNI metódus Java-ból egyszerűen hívható, de JSNI kódból is szükség lehet kliens oldali Java kód elérésére
- JSNI kód írásakor mi dolgunk a böngészőfüggetlenség megoldása

cross-site kérések:

- same origin policy megtiltja a böngészőknek, hogy más protokoll/szerver/port hármashoz AJAX kérést küldjenek, mint ahonnan az oldal letöltődött
- ez akkor probléma, ha a szerver oldali adatokat más szerverekről akarjuk lekérni
- megoldás:
  - **proxy**: ahonnan a GWT app letöltődik, ott egy proxy szolgáltatás áthív a többi szerverbe
  - JSON letöltése **<script> tag**gel, erre nem érvényes az SOP

Google Web Toolkit - GUI

alapelvek:

- a beépített GUI komponensek böngészőfüggetlenek
- igyekszik a böngésző natív UI elemeire támaszkodni
  - **panelek:** konténerek, melyekhez más panelek vagy egyéb widgetek adhatók hozzá
    - meghatározzák a gyermekelemek elhelyezését
    - a layout meghatározható deklaratívan, a host HTML-ben lévő HTML tagekkel is, a Panel osztályok segítségével viszont kódban adható meg
  - **widgetek:**
    - van kb. 20 beépített kiváló widget az utókornak, de fejleszthetsz sajátot is
  - vannak **editor widgetek** is, ami **GWT 2.1**-es újítás:
    - UI-ba írt adatok és memóriabeli objektumok közti adatkötést valósítják meg
    - tetszőleges JavaBeannel használhatók
    - tetszőlegesen kombinálhatók összetett widgetekben

eseménykezelés:

- mint kb. minden létező GUI keretrendszernél, itt is köthetünk handlert komponensekhez :)
- minden handler interfész **egy** metódust definiál, ebben **egy** GwtEvent gyerek a paraméter
- handlerek el is távolíthatók
- **GWT 1.5**-ig handlerek helyett listener interfészek voltak, de ez már deprecated
- kényelmes anonim osztályokban megvalósítani a handlereket, bár több memóriát fogyaszt
- alacsonyabb memóriahasználat érhető el több widgethez közös handlerrel, ebben egy getSource metódus mondja meg, honnan jött az event

a stílus teljesen CSS alapú, a CSS-t több módon be lehet húzni, vannak beépített témák, hudeizgi.

DOM manipuláció:

- GWT egyik fő előnye, hogy widgetek és panelek segítségével elfedi a böngészőfüggő DOM-ot, amit azért néha mégis el kell érni, pl.
  - saját widget fejlesztésénél,
  - host page-ben elhelyezett elem elérésénél,
  - böngészőesemények alacsony szintű kezelésénél
- erre egy eszköz a JSNI, de van Java alapú megoldás is: minden widget eléri az őt reprezentáló DOM elemet a `getElement()` metódussal

UiBinder: felhasználói felület deklaratív összeállítását teszi lehetővé **GWT 2.0 óta**

- **PRO:**
  - gyorsabb fejlesztés
  - megjelenítés és viselkedés szétválasztása
  - odaadható HTML fejlesztőknek, designereknek
  - gyorsan építhető, GUI-váz, ami fokozatosan bővíthető
  - jobb teljesítmény (innerHTML), saját I18N támogatás
- layoutra fókuszál, adatok HTML-é konvertálása továbbra is a widgetek feladata

- HTML megkötések:
    - nem minden widget fogad el törzsként HTML-t, erre a fejlesztőnek kell figyelnie
    - HTML entitások nem használhatók alapból ui.xml-ben, importálni kell őket :(

ClientBundle:

- sok kis kép letöltése negatív hatással van a teljesítményre:
    - a HTTP header overheadje nagy a törzshöz képest
    - több HTTP roundtrip történik cache-elés esetén is
    - a konkurens kapcsolatok korlátozása miatt az adatok AJAX-os letöltése késlekedik
- gyakori megoldás: képek elhelyezése egy nagy fájlban, CSS-ben ennek egy szeletére hivatkozunk, ezt csinálja a ClientBundle

I18N:

- Java properties fájl alapú
- de többet nyújt: le tudja generálni a properties fájlt, akár utasításokkal a fordítónak, bizonyos részeket placeholderekkel helyettesítve
- többféle támogatás: statikus, dinamikus, UiBinder-hez

history:

- egy GWT app alapvetően egyetlen oldal → böngésző back gomb hatására egy egész más oldalra fogunk jutni, nem az alkalmazás egy korábbi állapotára
- a fejlesztő feladata, hogy az app egyes állapotait egy history tokennel megjelölje
- a GWT ezt a tokent az URL fragment identifier részében fogja tárolni (# után)
- a fejlesztő feladata, hogy a history tokenből előállítson egy megfelelő alkalmazás-állapotot a History ValueChangeHandler-ében

biztonság:

- XSS veszélye fennáll közvetlen setHTML, setInnerHTML hívásoknál,
- megoldás, ha
    - mindenhol widgetet használunk, ahol lehet,
    - setInnerHTML helyett setInnerText-et hívunk,
    - safeHtml használata is segithet, de nem zár ki semmit igazából.

Portlet

portál – single point of interaction:

- nagyobb hatékonyság, minden szükséges információ, alkalmazás elérhető
- on demand
- bármilyen kliens eszközről
- perszonalizáció, testre szabhatóság
- aggregáció (egységes felület)
- single sign-on (egyszeri bejelentkezés)
- keresés
- együttműködés

alkalmazásszerver:

- web alapú és alkalmazás logikát implementáló komponensek futtató környezete, mely bizonyos alapszolgáltatásokat biztosít:
    - felhasználó- és jogosultságkezelés (authentikáció, authorizáció)
    - tranzakciókezelés
    - komponensek életciklus menedzsmentje
    - erőforrás pooling

portál szerver:

- alkalmazásszerverre épülő megoldáscsomag, mely a portál-alkalmazások futtató környezete
- tipikus alapszolgáltatásai:
  - felület aggregáció
  - perszonalizáció, testre szabott felületek
  - kliens-technológiák támogatása
  - skin-ezés
- további szolgáltatások:
  - dokumentumtár, content management
  - kollaborációs eszközök
  - RWW, wiki, blog, fórum
  - kész/testre szabható modulok

portletek:

- általában Java alapú webes komponensek, mely a portlet containerben fut, és a bejövő kérések feldolgozása által dinamikus tartalmat állít elő
- felhasználói felület komponensek, melyek a portálok építőelemei
- gyakran információs rendszerek felhasználói felület rétegét jelentik
- *felhasználó*: tartalmi és funkcionális egység, oldalakat felépítő elemek
- *admin*: telepíthető, konfigolható komponensek, jogosultsági beállítások rendelhetők hozzá
- *fejlesztő*: java kód, mely a portál szerveren fut, szolgáltatásra épít

portlet és servlet kapcsolata:

- specifikáció ajánlása szerint a portlet konténer szervlet konténerre épüljön
- **hasonlóság**: (a nevük :)))))))
  - szerver oldali java alapú webes komponens
  - konténervezérelt életciklus
  - dinamikus tartalmat generál
  - kérés-válasz alapú paradigma
- **eltérés**:
  - portletek a megjelenített oldalnak csak egy részét generálják
  - tovább finomított életciklus
  - portleteknél előre definiált üzemmódok és ablakállapotok
  - portletspecifikus konfiguráció perzisztens tárolása
  - eseménykezelés

portletek életciklusa:

- **inicializálás:**
  - *init*: konténer betölti és inicializálja a portletet
- **kérések kiszolgálása:**
  - *processAction*: vezérlőlogika végrehajtása
  - *render*: megjelenítéshez dinamikus tartalom generálása kliens felé
  - *serveResource*: erőforrásra vonatkozó kérés kiszolgálása
  - *processEvent*: portlethez érkező események kezelése
- **megsemmisítés:**
  - *destroy*: értesítés a konténertől a portlet megszüntetéséről – lefoglalt erőforrások felszabadítása

portletSession:

- adott klienshez tartozó kérés-válasz párok összefogása (basically = HttpServletSession)
- állapot tárolása a szervleteknél megszokott módon
- érvényességi tartománya lehet: APPLICATION_SCOPE vagy PORTLET_SCOPE
- konkurencia kezelésre figyelni kell :(

kliens oldali átirányítás:

- megfelelő HTTP fejlécek beállítása, átirányítás kliens böngészőjén
- paraméterként abszolút URL
- események küldése nem ajánlott

szerver oldali átirányítás:

- specifikáció támogatja vezérlés átadását szervleteknek és JSP-knek
- jó, de hogyan tudsz szerveroldalon átirányítani:
  - **include:** a külső erőforrás által generált tartalom beszúrása portlet kimenetébe
  - **forward:** vezérlés átadása egy külső erőforrásnak
- kontextus konverziójáról a konténer gondoskodik

cachelés támogatása:

- kliens oldali: kliens böngésző gyorsítótárban eltárolt oldalak URL alapján
- fajtái vannak neki:
  - **full**: a generált URL nem tartalmazza az oldal aktuális állapotát
  - **portlet**: biztosítja a hozzáférést a portlet állapotához
  - **page**: az oldal állapotától is függ a referált erőfottás tartalma ← **default**
- **szerveroldali gyorsítótár:**
  - szervleteknél szűrőt használtak, portleteknél szabványos megoldás
  - *deklaratív*: telepítési leíróban portletenként expiration és scope megadása
  - *programozott*: CacheControl objektumon állítasz expirationt / scopeot
  - expiration: cache lejárati ideje, 0 esetén nincs cachelés, -1nél sose jár le
  - scope: portletenként vagy megosztva felhasználók között
- **validáció alapú cachelés:**
  - cache lejárata esetén validációs token
  - cache érvényességének vizsgálata
    - érvénytelen cache esetén új validációs token és tartalom generálása
    - érvényesség esetén cache használata

portlet szűrők:

- kiszolgálási láncba szűrők illeszthetők be
- mind a beérkező kérés, mind a visszaküldött válasz manipulálható
- tipikus felhasználás: tömörítás, kódolás, cachelés, hozzáférés szabályozás
- kiszolgálási lánc megszakítható IS
- egyes kéréstípusokhoz külön szűrőtípusok (Action/Event/Render/ResourceFilter)

Web Services for Remote Portlets

A **WSRP** egy webszolgáltatás szabványt definiál, mely révén távol futó, felületet generáló (megjelenés-orientált) szolgáltatások, modulok integrálhatók online portál, illetve egyéb felületi alkalmazásba

- **producer:** felületi szolgáltatást nyújtó komponens, a producer jelenti a szolgáltatásként kiajánlott portletek / modulok futtató környezetét
- **consumer:** olyan köztes rendszer, mely a felhasználók nevében megjelenítés-orientált szolgáltatásokkal kommunikál

- **előnyök:**
  - a megjelenés és telepítés / futtatás helye egymástól elkülönül
  - a szolgáltatás nem csupán adatokat, és rajtuk végezhető műveleteket nyújt, de a megjelenítési logikát / vezérlést is – azt nem kell duplikáltan megvalósítani kliensben is
  - minimális fejlesztést / konfigurációt igényel
  - interoperabilitás, platformfüggetlenség (.NET consumer, Java provider)
  - széleskörű ipari támogatottság

Spring 5

Java 8:

- minimum kell hozzá, ki tudja használni a nyelvi lehetőségeit:
  - lambdák
  - default interface metódusok
  - jobb generikus típus következtetés
- az élet annyira szép, hogy még Java 9-el is teljesen kompatibilis :)
  - a framework buildelhező JDK 9-vel is, és a tesztek is átmennek azzal futtatva
  - legtöbb helyen kivették a JDK 9-ben deprecated API-kat

Java EE 7, Java EE 8 API-k:

- **minimum** a 7 meg van követelve
  - Servlet 3.1, Bean Validation 1.1, JPA 2.1, JMS 2.0
- kompatibilis futási időben Java EE 8-as API-kal is:
  - Servlet 4.0, Bean validation 2.0, JPA 2.2, JSON Binding API 1.0

Spring core módosítások:

- beanek keresésének optimalizálása:
  - már fordítási időben scannel és a META-INF/spring.components fájlba generál
  - `spring-context-indexer` függőség kell hozzá
  - a szokásos annotációkon kívül a javax-os vagy `@Index` annotációval ellátott típusokat is beleteszi
- opcionális injektálási pont:
  - ha nem talál megfelelő beant, nem dob kivételt, hanem null marad
  - `Optional<Bean> bean` // Spring 4 óta
  - `@Nullable Bean bean` // Spring 5-ös lehetőség
- `@Nullable` egyéb lehetőségei:
  - nullt elfogadó argumentumok, nullal visszatérni tudó metódusok megjelölésére → fordítási időben detektálhatók NullPoinetExceptionök
- néhány Spring API már nem fogad el nulloakt (pl. `StringUtils`)
- saját commocs logging bridge (autodetektálja a log4j2-t, SLF4J-t, java.util.loggingot)
- resource interfészen isFile metódus (eddig csak a getFile kivételéből derült ki)

reaktív programozás:

- olyan programozói modell, ahol a kódunk változásokra reagál, pl.
  - hálózati komponens IO eseményekre, UI controller komponens egér eseményekre
- webes környezetben a kérések konkurens feldolgozására két fő lehetőség:
  - minden konkurens kérésnek saját szál (→ sok szál, rossz skálázódás)
  - egy szál több kérést is kezel egyszerre, azáltal, hogy kis IO csomagokat küld/fogad több hálózati kapcsolaton, miközben feldolgozást is végez
    - → reaktív szemlélet, IO eseményekre reagál a kódunk (jó skálázódás, de csak akkor, ha a szál nem blokkolódik → minden aszinkron kell legyen)

    - aszinkron programozás callback metódusokkal callback hellhez vezethez
      → a reaktív programozás ezt kiküszöböli
    - külön kezelendő a túlterhelés:
        - blokkoló esetben a kliensek nem tudnak következő hívást indítani, míg nem készült el a válasz → természetes visszajelzés
        - aszinkron esetben külön kell róla gondoskodni
    - **kapcsolódó Java API-k**:
        - RxJava: egy népszerű reaktív library, ReactiveX Java megvalósítása
        - ReactiveStreams: egységes API specifikáció aszinkron streamek reaktív feldolgozására, megvalósítására (pl. Reactor, RxJava 2.0, …, Java 9 is átvette, bye)
        - Spring 5 a Reactor-ra épít!

Reaktív adatréteg:

- reaktív szemlélet előnyei akkor használhatók ki teljes mértékben, ha az adatelérés is reaktív
- ehhez feltétel az aszinkron működésű adatbázis driver:
    - relációs adatbázisokhoz még nincs, de már dolgoznak rajta
    - MongoDB, Couchbase, Cassandra, ezekhez van → Spring Data 2.0 ad hozzá reaktív API-t

WebFlux ↔ Spring Web MVC:

- webflux nem minden alkalmazásban praktikus
    - a reaktív programozást meg kell szokni
    - webflux teljesítmény előnye IO intenzív esetekben érezhető igazán
    - ha az adatréteg amúgy is blokkol, lehet, hogy kicsi a nyereség
- alapelv: ha nincs igazoltan webrétegbeli blokkolásból, szálkezelésből adódó teljesítményprobléma, nem érdemes váltani

Spring Web MVC frissítések:

- HTTP entitások kezelésére library frissítések:
    - Jackson 2.9
    - Protocol Buffers 3.0
    - Java EE 8 JSONB
- reaktív támogatás:
    - controller metódusok visszatérési értéke lehet Ractor 3.1 vagy RxJava 1.3/2.1 reaktív típusa (Flux, Mono, Observable)
    - reaktív klienst itt is használhatunk más szolgáltatások meghívására
    - de a servlet API-ban már blokkolódik a válasz írása! (webfluxnál még nem)

HTTP/2:

- HTTP/2 Server Push használható
- Spring 5.1 óta a Servlet 4.0-s PushBuilder használható, controller metódus argumentumként injektálható
- de a Spring 5.0-ban is használható natív Jetty 9.3/Tomcat 9/Undertow 1.4 API-t használva

kotlin támogatás:

- JetBrains által kifejlesztett JVM nyelv
    - támogatja a funkcionális programozást
    - invariáns tömbök
    - nincsenek nyers típusok
    - nullreferenciák compiler szintű ellenőrzése
    - nincsenek benne checked exception-ök
    - string templatek
- Spring 4-ben is haszálható, de az 5 kifejezetten támogatja

Spring test bővítések:

- JUnit5-ös @EnabledIf, @DisabledIf elfogad SpEL-t
- Spring TestContext párhuzamosan tud teszteket futtatni
- TestRestTemplate mintájára reaktív WebTestClient

törölt API-k, nem támogatott technológiák:

- NeanFactoryLocator
- jdbc.support.nativejdbc
- Tiles2, Hibernate3/4
- Portlet
- Velocity
- JasperReports
- XMLBeans
- JDO
- Guava