JEGYZŐKÖNYV

Webes adatkezelő környezetek

Féléves feladat

Szállásfoglalási rendszer

Készítette: **Vaszil Valentin**
Neptunkód: **XGBT95**
Dátum: **2025. december**

Miskolc, 2025

Tartalomjegyzék

Bevezetés

A webes adatkezelő környezetek tantárgy féléves feladata egy olyan összetett, több lépésből álló projekt megvalósítása, amelynek célja a strukturált adatreprezentáció, az XML alapú adatkezelés, valamint a DOM feldolgozási technikák gyakorlati elsajátítása. A feladat általam választott témája egy szállásfoglalási rendszer megtervezése és kivitelezése, amely a valós életben is gyakran előforduló problémára ad modell szintű megoldást. A projekt során a relációs adatbázisokban megszokott tervezési lépések XML környezetre való átültítése áll a középpontban: az ER modell készítésétől kezdve az XDM modell kialakításán át egészen az XML dokumentum és az XML séma megalkotásáig. A cél, hogy a rendszer entitásai, kapcsolatai és tulajdonságai átlátható és formalizált módon jelenjenek meg a különböző reprezentációs szinteken.
A feladat első része az adatmodellezésről szól, ahol meg kell határozni a rendszer működéséhez szükséges egyedeket, azok kapcsolatait és a hozzájuk tartozó kulcsokat. Az ER modell létrehozása után ezt át kell alakítani XDM modellé, mely már az XML struktúráját követi. A következő lépés az XML dokumentum elkészítése a megfelelő hierarchiával, ismétlődő elemekkel, attribútumokkal és kommentekkel. Az adatok érvényességét az XML Schema biztosítja, amelyben a saját egyszerű és komplex típusok, a kulcsok, az idegen kulcsok, valamint a ref hivatkozások is definiálásra kerülnek. Ez a lépés különösen fontos, hiszen az XSD adja meg az XML dokumentum formai és szerkezeti szabályait.
A projekt második része a Java DOM technológia alkalmazására épül. Ennek keretében a létrehozott XML dokumentumot programból kell beolvasni, kiírni, lekérdezni és módosítani. A feladat a DOM fa bejárását, az elemek és attribútumok elérését, valamint az új csomópontok létrehozásának és módosításának gyakorlatát is magában foglalja.

1. Az első feladat címe

Az adatbázis megtervezése, ER, XDM modellek megvalósítása. XML és XSD fájlok készítése.

1.1 Az adatbázis ER modell megtervezése

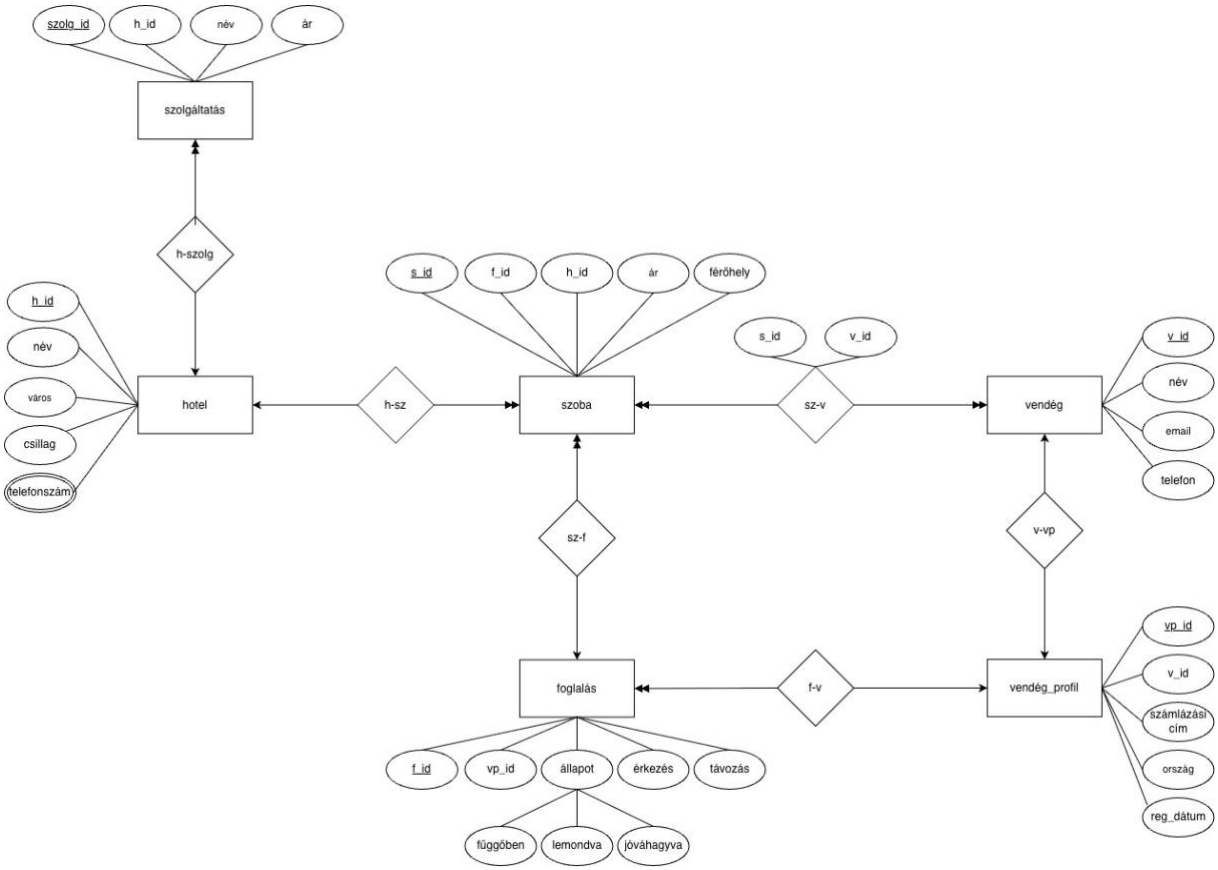
Az ER modell elkészítésénél az volt a célom, hogy a szállásfoglalási rendszer működését logikailag pontosan, egyértelműen és áttekinthetően ábrázoljam. Mivel a rendszer több különböző típusú szereplőt és kapcsolatot tartalmaz, első lépésként az egyedeket és azok legfontosabb tulajdonságait határoztam meg. Ezekből indult ki a kapcsolatstruktúra is, amely végül többféle kapcsolatot foglal magában: 1:1, 1:N és M:N összeköttetéseket is.

Egyedek (kulcsok és főbb attribútumok)

- Hotel (h_id — PK)
    - név
    - város
    - telefonszám
    - csillag (besorolás)
- Szoba (s_id — PK)
    - férőhely
    - ár
    - h_id (FK → Hotel)
    - f_id
- Szolgáltatás (szolg_id — PK)
    - név

    - ár
    - h_id (FK → Hotel)
- Vendég (v_id — PK)
    - név
    - email
    - telefon
- Vendég_profil (vp_id — PK)
    - v_id (FK → Vendég)
    - számlázási cím
    - ország
    - reg_dátum
- Szoba_vendég
    - v_id
    - sz_id
- Foglalás (f_id — PK)
    - vp_id (FK → Vendég_profil)
    - érkezés
    - távozás
    - állapot (összetett/többértékű: függőben, lemondva, jóváhagyva)

Az elkészült ER modellt szabványos jelölésekkel rajzoltam meg, külön figyelve arra, hogy az egyedek, tulajdonságok és kapcsolatok jól elkülönüljenek, és a diagram áttekinthető maradjon. A kapcsolatok nevei, valamint a kulcsok és összetett kulcsok jelölése is egyértelműen szerepel a modellen. A végeredmény egy olyan logikai adatmodell lett, amely teljesen lefedi a rendszer működését, és megfelelő alapot ad a további lépésekhez, különösen az XDM modell és később az XML dokumentum elkészítéséhez.



1. ábra: A szállásfoglalási rendszer ER modellje

1.2 Az ER modell konvertálása XDM modellre

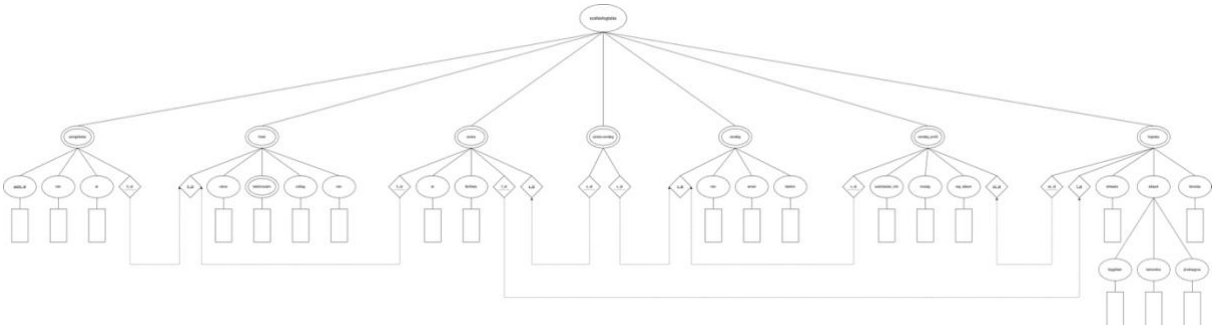
Miután elkészült az ER modell, a következő feladatom az volt, hogy ezt XML-orientált formára, vagyis XDM modellre alakítsam át. Az ER modell relációs logikája és az XML hierarchikus felépítése között lényeges különbségek vannak, ezért a konverzió során végig arra figyeltem, hogy minden egyed és kapcsolat értelmesen átültethető legyen az XML struktúrájába. Az XDM modell tulajdonképpen az XML dokumentum szerkezeti vázát jelenti, így a rendszer egész felépítését már ebben a formában kellett újragondolni.

Első lépésként meghatároztam a gyökérelemet, amely a teljes dokumentumot összefogja. Ez a **szallasfoglalas** lett, és minden más elem ehhez a gyökérhez kapcsolódik. Ennek megfelelően a modellben a hotel, a szolgáltatás, a szoba, a vendég, a vendégprofil és a foglalás mind közvetlenül a gyökérelem alá került. Az XML-ben a sorrend nem követelmény, de a szemléletesség kedvéért logikusan csoportosítottam az elemeket.

Az ER modellben szereplő kapcsolatokat XML-ben másképp kellett kifejeznem. Relációs környezetben a kapcsolatokat kapcsolatvonalak jelölik, XML-ben viszont nem létezik ilyen fogalom, ezért az összeköttetéseket **azonosítók és hivatkozások** segítségével kellett felépíteni. Minden egyed megkapta a saját azonosítóját (például h_id, s_id, v_id), és ezekre támaszkodva kerültek kialakításra az idegen kulcs jellegű attribútumok. Például a szobánál a h_id az adott hotelhez kapcsolja az elemet, a foglalás pedig a vp_id révén kapcsolódik a vendégprofilhoz. Így a relációs adatbázisból ismert 1:N és 1:1 kapcsolatok XML-ben attribútum alapú összekötésekké alakultak.

A **M:N kapcsolatot**, amely a szoba és a vendég között jelent meg, egy önálló elembe kellett áthelyeznem, hasonlóan egy kapcsolótáblához. Ez lett a **szoba-vendeg** elem, amely két attribútummal (s_id és v_id) valósítja meg a kapcsolatot. Ez az XDM modellben is külön elemként jelenik meg, és megteremti a lehetőséget arra, hogy egy szobához több vendég is kapcsolódjon, illetve egy vendég több szobában is szerepelhessen.

Az ER modell többi eleme is az XDM sajátosságainak megfelelően került átalakításra. A vendégprofil például egyértelműen illeszkedik az XML szerkezetébe, hiszen természetesen felírható hierarchiában: a számlázási cím, az ország és a regisztráció dátuma mind al-elemként jelenik meg. A többi entitás felépítése is jól illeszthető volt az XML-re jellemző tagekhez és attribútumokhoz.
Az elkészült XDM modellt szabványos jelölésekkel rajzoltam meg, ügyelve arra, hogy a PK–FK kapcsolatok ne keresztezzék egymást, és hogy a hivatkozások egyértelműen követhetőek legyenek. A modell így már teljes mértékben az XML gondolkodásmódjára épül, és gyakorlatilag az XML dokumentum szerkezeti terveként szolgált.



2. ábra: A szállásfoglalási rendszer XDM modellje

1.3 Az XDM modell alapján XML dokumentum készítése

Az XDM modell elkészítése után a következő lépés az volt, hogy a megtervezett struktúrát konkrét XML dokumentummá alakítsam. Ennél a résznél az volt a fő szempontom, hogy az XML szerkezete teljes mértékben kövesse az XDM modellt, és minden elem pontosan abban a formában jelenjen meg, ahogyan a modell előírja. Mivel az XML dokumentum később a DOM feldolgozás alapját is adja, különösen fontos volt, hogy jól áttekinthető, tagolt, és könnyen bejárható legyen.
Először létrehoztam a gyökérelemet, amely a **szallasfoglalas** nevet kapta, ugyanúgy, ahogyan az XDM modellben is szerepel. Ide kerültek be a rendszer főbb elemei: a hotel, a szolgáltatás, a szoba, a szoba–vendég kapcsolat, a vendég, a vendégprofil és a foglalás. Ezeket egymás után helyeztem el, logikai csoportosítás szerint elkülönítve. Arra is figyeltem, hogy minden olyan elem, amely többször előfordulhat, valóban több példányban szerepeljen. Így például két hotel, két szolgáltatás, két szoba, két vendég és két foglalás is bekerült a dokumentumba.
Az XML felépítésénél törekedtem arra, hogy minden elem jól olvasható legyen. Ezt segíti a megfelelő behúzás, a tagolt struktúra, valamint a magyarázó megjegyzések, amelyeket a dokumentumba is beillesztettem. Ezek különösen hasznosak akkor, ha valaki először találkozik az XML szerkezetével, vagy a későbbi feldolgozás során szeretném gyorsan visszakeresni, hogy melyik rész mit tartalmaz. Az attribútumokat szintén az XDM modell szerint helyeztem el, például a Hotel esetében a h_id, a Szoba esetében pedig az s_id, h_id és f_id attribútumok kerültek elő.
A tartalmi kitöltésnél igyekeztem valósághű adatokat használni, hogy a dokumentum ne csak szerkezetileg, hanem tartalmilag is értelmezhető legyen. A hotelek, szolgáltatások és foglalások valósághű mintákat követnek, és a vendégek adatai is olyan formában szerepelnek, hogy a Java programom, későbbi feldolgozás során könnyen tudja azt olvasni.

XML kódrészlet:

```
<!-- Foglalások -->

<!-- Foglalás 1 -->
```

```
<foglalas f_id="1" vp_id="1">
  <erkezes>2025-02-10</erkezes>
  <tavozas>2025-02-15</tavozas>
  <allapot>
    <fuggoben>false</fuggoben>
    <lemondva>false</lemondva>
    <jovahagyva>true</jovahagyva>
  </allapot>
</foglalas>

<!-- Foglalás 2 -->
<foglalas f_id="2" vp_id="2">
  <erkezes>2025-03-01</erkezes>
  <tavozas>2025-03-05</tavozas>
  <allapot>
    <fuggoben>true</fuggoben>
    <lemondva>false</lemondva>
    <jovahagyva>false</jovahagyva>
  </allapot>
</foglalas>
```

Teljes forráskód: XGBT95_XML.xml

1.4 Az XML dokumentum alapján XML Schema készítése

Miután elkészült az XML dokumentum, a következő feladatom az volt, hogy ehhez létrehozzam a megfelelő XML sémafájlt (XSD). A séma célja az, hogy meghatározza az XML dokumentum szerkezetét, szabályait és az elemek közötti kapcsolatokat. Fontos volt, hogy az XSD ne csak egyszerű típusdefiníciókat tartalmazzon, hanem a teljes adatstruktúrát pontosan és szigorúan leírja. Emiatt nagy hangsúlyt fektettem a saját típusok kialakítására, a komplex típusok megtervezésére, valamint a kulcs–idegen kulcs hivatkozások definiálására is.

Első lépésként létrehoztam a saját **egyszerű típusokat**, amelyekkel az ismétlődő adatkorlátozásokat tudtam egységesen kezelni. Ilyen volt például a pozitív egész számot leíró típus, a pénzösszeghez tartozó decimal alapú típus, vagy a maximum 50 karakteres szöveg. Ezek azért fontosak, mert az XML dokumentumban rengeteg olyan mező van, ahol ezek a korlátozások ismétlődnek, és így a sémában átlátható módon, újrahasznosítható formában tudtam ezeket kezelni.

Ezután következtek a **komplex típusok**, amelyekkel az egyedek teljes szerkezetét határoztam meg. Minden entitás (hotel, szoba, szolgáltatás, vendég, vendégprofil, foglalás) saját komplex típust kapott, benne az al-elemekkel és a szükséges attribútumokkal. Arra törekedtem, hogy a típusok felépítése a lehető legjobban tükrözze az XDM és XML dokumentum struktúráját, így az XSD-ből egyértelműen látszik, hogy milyen elemek és milyen hierarchiában épülnek egymásra.

A séma fontos része volt a **ref használata**, amely lehetővé tette, hogy bizonyos elemeket globálisan definiáljak, majd a gyökérelem belsejében már ref hivatkozással helyezzem el őket. Ez sokkal átláthatóbbá tette a dokumentumot, mert nem kellett minden elemet külön-külön újra definiálni, csak hivatkoztam a típusára. Emellett ebből a megközelítésből a kulcsdefiníciók kezelése is rendezettebb lett.

Az XSD-ben a legnagyobb hangsúlyt a **key** és **keyref** elemekre helyeztem. Ezekkel tudtam leírni a különböző egyedek közötti kapcsolatokat, hasonlóan ahhoz, ahogyan az ER modellben a kulcsok és az idegen kulcsok működnek. A hotel, a szoba, a vendég, a vendégprofil és a foglalás mind saját elsődleges kulcsot kapott a sémafájlban. A kapcsolatok (például a szoba–hotel, vendégprofil–foglalás vagy a szoba–vendég kapcsolótábla) pedig keyref segítségével kerültek megvalósításra. Ez biztosítja, hogy az XML dokumentumban nem lehetnek érvénytelen hivatkozások, például olyan szoba, amely nem létező hotelhez tartozik.
A kapcsolótábla jellegű „szoba-vendeg" elem esetében külön odafigyeltem arra, hogy a két attribútum együtt alkot egy összetett elsődleges kulcsot. Ezt az XSD-ben is tükrözni kellett, ezért itt egy több mezőből álló key definíció készült, amely egyedivé teszi a s_id – v_id párosokat.

XSD kódrészlet:

```
<!-- ======================================== -->
<!-- Globális elemek a ref használatához -->
<!-- ======================================== -->

<xs:element name="hotel" type="HotelTipus"/>
<xs:element name="szolgaltatas" type="SzolgaltatasTipus"/>
<xs:element name="szoba" type="SzobaTipus"/>
<xs:element name="szoba-vendeg" type="SzobaVendegKapcsolatTipus"/>
<xs:element name="vendeg" type="VendegTipus"/>
<xs:element name="vendeg_profil" type="VendegProfilTipus"/>
<xs:element name="foglalas" type="FoglalasTipus"/>
```

Teljes forráskód: XGBT95_XMLSchema.xsd

2. A második feladat leírása

A második feladat célja az XML dokumentum programozott feldolgozása Java nyelven, a DOM (Document Object Model) technológia alkalmazásával. A megvalósításhoz három osztályt kellett létrehozni: **DOMRead**, **DOMQuery** és **DOMModify**, amelyek külön feladatkörök szerint kezelik az XML dokumentumot.
A **DOMRead** osztály az XML fájl beolvasását, a dokumentum teljes tartalmának áttekinthető formában történő kiíratását, valamint a konzolra megjelenített kimenet fájlba mentését végzi.
A **DOMQuery** osztály különböző lekérdezések megvalósítására szolgál. A lekérdezések DOM-csomópontok bejárásával történnek, XPath használata nélkül, és a dokumentum különböző elemeire, attribútumaira vagy kapcsolataira irányulnak.
A **DOMModify** osztály feladata az XML dokumentum szerkezeti vagy adattartalmi módosítása. A módosítások végrehajtása után az eredmény a konzolon is megjelenítésre kerül, így követhetővé válik a dokumentum frissített állapota.
A három osztály együtt lefedi az XML feldolgozásának alapvető DOM-műveleteit: az olvasást, a fa bejárását, a lekérdezéseket és a módosítást.

2.1 Adatolvasás

Az adatolvasás megvalósítása a *DOMRead* osztály feladata. A program az XML dokumentum beolvasásához a Java beépített DOM API-ját használja. A DocumentBuilderFactory és DocumentBuilder objektumok segítségével betölti a **XGBT95_XML.xml** fájlt, majd normalizálja a dokumentumot. A normalizálás biztosítja, hogy a DOM fa szerkezete egységes és könnyen bejárható legyen.
A program ezután végigiterál a gyökérelem közvetlen gyermekelemein, és minden elemhez kiírja:

- az elem nevét,
- az összes attribútumát,
- a hozzá tartozó közvetlen al-elemek nevét és szövegét.

A kiírt szöveg egy StringBuilder objektumban gyűlik össze, amely később a konzolra és egy külön fájlba is elmentésre kerül.

Az alábbi részlet mutatja be a “hotel” gyerekelem közvetlen feldolgozását:

```
NodeList hotelList = doc.getElementsByTagName("hotel");

 out.append("\n\n=== Hotelek ===\n");

 for (int i = 0; i < hotelList.getLength(); i++) {

    Element e = (Element) hotelList.item(i);

    out.append("\nHotel ID:              ").append(e.getAttribute("h_id")).append("\n");

    out.append("Név: ")
.append(e.getElementsByTagName("nev").item(0).getTextContent()).append("\n");

    out.append("Város: ")

.append(e.getElementsByTagName("varos").item(0).getTextContent()).append("\n");

    NodeList tels = e.getElementsByTagName("telefonszam");

    out.append("Telefonszámok:\n");

    for (int t = 0; t < tels.getLength(); t++) {

       out.append("  -   ").append(tels.item(t).getTextContent()).append("\n");

    }

    out.append("Csillag: ")
.append(e.getElementsByTagName("csillag").item(0).getTextContent())

    .append("\n");

}
```

Magyarázat:
Ez a ciklus bejárja a hotel elemet. Majd az előre meghatározott al-elem nevek szerint az adatokat a kimeneti változóba, formázottan menti. A kimenet ezáltal áttekinthető, hierarchikus formában mutatja be az XML dokumentum tartalmát.

A konzolra írt kimenet fájlba mentése
A program a kiírt eredményt egy egyszerű szövegfájlba is elmenti:

```
FileWriter fw = new FileWriter("XGBT95_read_output.txt");
fw.write(output.toString());
fw.close();
```

A teljes forráskód: XGBT95DOMRead.java

2.2 Adat-lekérdezés

Az adat-lekérdezést a *DOMQuery* osztály valósítja meg. A program célja, hogy különböző információkat nyerjen ki az XML dokumentumból, a DOM fa bejárásával, XPath használata nélkül. A program a dokumentum beolvasása és normalizálása után négy különálló lekérdezést hajt végre, amelyek az XML eltérő részeire fókuszálnak.

A lekérdezésekhez a getElementsByTagName() metódust alkalmazza, amely segítségével az adott elemcsoport NodeList formájában elérhető. A program ezután minden esetben végigiterál a találati listán, és a megfelelő adatokat a konzolra írja.

1. lekérdezés – Hotelek neve és városa

Az első lekérdezés az összes hotel nevét és városát jeleníti meg. A program minden *hotel* elem alatti nev és varos al-elemet kiolvas:

```
NodeList hotels = doc.getElementsByTagName("hotel");
for (int i = 0; i < hotels.getLength(); i++) {
   Element h = (Element) hotels.item(i);
   String nev = h.getElementsByTagName("nev").item(0).getTextContent();
   String varos = h.getElementsByTagName("varos").item(0).getTextContent();
   System.out.println("- " + nev + " (" + varos + ")");
}
```

Magyarázat:
A metódus minden hotel elemre külön lekéri a gyermekelemek szövegértékét, majd formázott sorban jeleníti meg.

2. lekérdezés – Ötcsillagos hotelek
A második lekérdezés az 5 csillagos hoteleket listázza. A csillag elem szöveges tartalma alapján történik a szűrés:

```
if (h.getElementsByTagName("csillag").item(0).getTextContent().equals("5")) {
   System.out.println("- " + h.getElementsByTagName("nev").item(0).getTextContent());
```

```
}
```

Magyarázat:
A program minden hotel esetében ellenőrzi, hogy a csillagok száma 5-e, és csak ezek nevét írja ki.

3. lekérdezés – Vendégek nevei
A harmadik lekérdezés az összes *vendeg* elem nevét jeleníti meg:

```
NodeList guests = doc.getElementsByTagName("vendeg");
for (int i = 0; i < guests.getLength(); i++) {
   Element v = (Element) guests.item(i);
   System.out.println("- " + v.getElementsByTagName("nev").item(0).getTextContent());
}
```

Magyarázat:
A megoldás egyszerű fa-bejárással listázza ki minden vendég nevét, további szűrés nélkül.

4. lekérdezés – 3 vagy több férőhelyes szobák
A negyedik lekérdezés azoknak a szobáknak az azonosítóját írja ki, amelyek férőhelye legalább 3:

```
NodeList rooms = doc.getElementsByTagName("szoba");
for (int i = 0; i < rooms.getLength(); i++) {
   Element s = (Element) rooms.item(i);
   int ferohely = Integer.parseInt(s.getElementsByTagName("ferohely").item(0).getTextContent());
   if (ferohely >= 3) {
      System.out.println("- Szoba ID: " + s.getAttribute("s_id")
                  + " (férőhely: " + ferohely + ")");
   }
}
```

Magyarázat:
A program numerikus összehasonlítást alkalmaz a férőhely alapján, és csak a feltételnek megfelelő szobákat jeleníti meg.

A teljes forráskód: XGBT95DOMQuery.java

2.3 Adatmódosítás

Az adatmódosítást a *DOMModify* osztály valósítja meg. A program a már ismert módon beolvassa és normalizálja az XML dokumentumot, majd több különböző szerkezeti és adattartalmi módosítást hajt végre rajta. A változtatások mind a DOM fa közvetlen manipulálásával történnek, a módosított dokumentum pedig a konzolra kerül kiírásra.
A módosítások négy lépésben történnek, az alábbiak szerint.

1. módosítás – Hotel csillagszámának átírása
A program lekéri az első *hotel* elemet, és módosítja a csillag al-elem szövegértékét:

```
Element hotel1 = (Element) doc.getElementsByTagName("hotel").item(0);
hotel1.getElementsByTagName("csillag").item(0).setTextContent("3");
```

Magyarázat:
A DOM fa megfelelő csomópontjához közvetlenül hozzáférve egyszerű szövegcserével frissül a csillagszám értéke.

2. módosítás – Új szolgáltatás hozzáadása
A program egy teljesen új szolgaltatas elemet hoz létre, attribútumokkal és al-elemekkel, majd ezt a gyökérelemhez fűzi:

```
Element newServ = doc.createElement("szolgaltatas");
newServ.setAttribute("szolg_id", "3");
newServ.setAttribute("h_id", "1");

Element nev = doc.createElement("nev");
nev.setTextContent("Parkolás");
Element ar = doc.createElement("ar");
ar.setTextContent("2500");

newServ.appendChild(nev);
newServ.appendChild(ar);

Comment komment = doc.createComment(" Szolgáltatás 3 ");
Text uresSor = doc.createTextNode("\n    ");

NodeList szolgList = doc.getElementsByTagName("szolgaltatas");
Node uSzolg = szolgList.item(szolgList.getLength() - 1);
Node kovNode = uSzolg.getNextSibling();
gyoker.insertBefore(uresSor, kovNode);
gyoker.insertBefore(komment, kovNode);
gyoker.insertBefore(uresSor.cloneNode(false), kovNode);
gyoker.insertBefore(newServ, kovNode);

System.out.println("Új szolgáltatás hozzáadva (Parkolás).");
```

Magyarázat:
A DOM API használatával új elemek és attribútumok dinamikusan hozhatók létre, majd a dokumentum tetszőleges részéhez hozzáfűzhetők.

3. módosítás – Vendég email címének cseréje
A program módosítja az első *vendeg* elem email al-elemeinek tartalmát:

```
Element vendeg1 = (Element) doc.getElementsByTagName("vendeg").item(0);
vendeg1.getElementsByTagName("email").item(0).setTextContent("uj.email@example.com");
```

Magyarázat:
Egyszerű értékcsere történik a gyermekcsomópont szövegében, így az új email cím azonnal megjelenik a DOM fában.

4. módosítás – Foglalás állapotának frissítése
A program az első *foglalas* elem állapotát módosítja úgy, hogy a jovahagyva elem értéke „false”, lemondva pedig “true” legyen:

```
Element foglalas1 = (Element) doc.getElementsByTagName("foglalas").item(0);
Element allapot = (Element) foglalas1.getElementsByTagName("allapot").item(0);
allapot.getElementsByTagName("lemondva").item(0).setTextContent("true");
allapot.getElementsByTagName("jovahagyva").item(0).setTextContent("false");
System.out.println("Foglalás #1 lemondva értéke true-ra állítva.");
```

Magyarázat:
A többszintű hierarchiában elhelyezkedő elem közvetlen eléréssel és értékcserével módosul.

A teljes forráskód: XGBT95DOMModify.java