BME Villamosmérnöki és
Informatikai Kar

Távközlési és Médiainformatikai Tanszék
Adatbázisok Labor

ADATBÁZISOK LABORATÓRIUM

Oktatási segédanyag az Adatbázisok laboratórium tantárgyhoz

Egyetemi belső használatra

18. javított kiadás

szerkesztette:
Gajdos Sándor

szerzők:
Balázs Zoltán
Erős Levente
Gajdos Sándor
Golda Bence
Győr Ferenc
Hajnács Zoltán
Hunyadi Levente
Kardkovács Zsolt
Kollár Ádám
Mátéfi Gergely
Marton József
Nagy Gábor
Nagypál Gábor
Paksy Patrik
Sallai Tamás
Soproni Péter
Unghváry Ferenc
Varsányi Márton
Veres-Szentkirályi András

2022.

Tartalom

Előszó

Ez a segédanyag (a továbbiakban: Segédlet) a BME Villamosmérnöki és Informatikai karán folyó műszaki informatikus alapképzés (BSc) tantervében a 3. szemeszterben szereplő "Adatbázisok" c. tantárgy laboratóriumi foglalkozásaihoz készült. Ez hivatott arra, hogy az "Adatbázisok" c. elméleti tantárgyhoz gyakorlati ismereteket is szolgáltasson. A Segédlet a tárgyhoz kapcsolódó, összesen 6[1] db foglalkozás segédanyagául szolgál.

A laboratóriumi foglalkozások keretében természetesen csak arra nyílik lehetőség, hogy az elméletben megtanultaknak egy viszonylag szűk részével találkozzanak a hallgatók: ez a rész a relációs adatbázis-kezelés és alkalmazásfejlesztés egyes elemeit jelenti. Eközben erősen építünk azokra az ismeretekre, amelyeket pl. az Adatbázisok tárgy előadásain lehet megszerezni a relációs adatmodellel, adatmodellezéssel, relációs sématervezéssel, lekérdező nyelvekkel, tranzakciókezeléssel kapcsolatban. A laboratóriumok anyagába bekerülő elemek folyamatosan változtak és változnak, ahogyan az adatbázis-kezelés súlypontjai is eltolódnak. Ennek megfelelően az 1998-as évtől kezdődően a Java nyelv és az adatbázisok kapcsolatát célzó labor, 2002-től dinamikus weboldalak készítése PHP nyelven jelent meg újdonságként a tematikában, 2003-tól az XML alapú adatkezelés és alkalmazásfejlesztés egyes lehetőségeivel ismertetjük meg a hallgatóságot, 2012-től pedig egy példát mutatunk arra, hogyan lehet SOA elvek mentén működő, kliens-szerver architektúrájú rendszert készíteni relációs adatbázisok tartalmának hozzáférésére/hozzáférésével. Célunk elsődlegesen nem készségszintű ismeretek nyújtása, sokkal inkább a fontosabb, adatbázis-kezeléssel kapcsolatos technológiákkal való gyakorlati ismerkedés lehetőségének biztosítása.

A 2017-től bevezetett „kontakt óraszámcsökkentés" következtében megváltoztak a tárgy oktatásának lehetőségei. A korábbi 6. félévről a 3. – az elméleti előadásokkal és tantermi gyakorlatokkal azonos – félévre került laborok már csak a relációs adatbáziskezeléshez legszorosabban kapcsolódó technológiai elemek bemutatását teszik lehetővé (kereskedelmi relációs adatbáziskezelő felépítése, SQL nyelv + alapszintű lekérdezés optimalizálás). Ugyanakkor az érdeklődők számára valamint referenciaként a Segédletben benne hagytuk a korábbi években a labor keretében tanítottakhoz tartozó anyagokat is. A visszajelzések alapján ezeket a gyakorló alkalmazásfejlesztők szívesen forgatják.

A laboratóriumi foglalkozások az Oracle relációs adatbázis-kezelő rendszer (18cR1 verziója) segítségével valósulnak meg. Tekintettel az Oracle szerteágazó lehetőségeire meg sem kíséreltük, hogy kimerítő ismereteket próbáljunk meg átadni a rendszerről a laboratóriumok és a Segédlet erősen korlátozott lehetőségei között.

Bár a szerzők és a szerkesztő mindent elkövettek, hogy a Segédletben leírtak pontosak, korrektek legyenek, ennek ellenére előfordulhatnak benne hibák. Ebben az esetben a visszajelzéseket köszönettel vesszük – sőt kérjük – a gajdos@db.bme.hu címre eljuttatni.

Budapest, 2022. október

Gajdos Sándor
Távközlési és Médiainformatikai Tanszék
Adatbázisok Labor

[1] Az oktatási szünetek miatt elmaradó foglalkozások miatt nem mindig fér bele egy félév programjába valamennyi.

I. labor: Az Oracle adatbázis-kezelő

Szerzők: Gajdos Sándor, Kardkovács Zsolt, Győr Ferenc, Marton József

„I am Sir Oracle,
And when I ope my lips, let no dog bark!"
(Shakespeare: The Merchant of Venice)

1. Miért éppen az Oracle?

Az adatbázis-kezelők piacán széles választékban állnak rendelkezésre a különböző hatékonyságú és technológiájú eszközök. A labor célja, hogy megismertessük a hallgatóval a korszerű adatbázis-kezelés néhány fontosabb elemét, ugyanakkor a lehetőségekhez képest naprakész tudással is felruházzuk őt. Ebből a célból a magyar piacon nagymértékben domináló terméket, az Oracle Database adatbázis-kezelőt (a továbbiakban, röviden: Oracle) mutatjuk be részletesebben. Megemlítjük – a teljesség igénye nélkül –, hogy az ismertebb és támogatottabb adatbázis-kezelők (továbbá gyártóik) a következők: *SQL Server* (relációs, Microsoft), *DB/2*, *Informix* (relációs, IBM), *Ingres II*[2] (relációs, Actian Corporation, GNU GPL), *Sybase Adaptive Server* (relációs, Sybase), *MySQL*[3] (relációs, Oracle Corporation, GNU GPL), *O2* (objektumorientált, O2 Inc.), *GemStone* (objektumorientált, GemTalk Systems), *ObjectStore* (objektumorientált, ObjectStore Corporation, Versata csoport).
A későbbiekben az adatbázis-kezelők további érdekesebb alkalmazásaival az ez iránt érdeklődők például az Adatbázisok elmélete c. MSc tárgy keretében találkozhatnak.
A laborokon az Oracle 18cR1 verziójú[4] szerverével valamint Java platformon futó egyes kliens eszközeivel fogunk dolgozni.

[2] A University of Californián kezdődött az Ingres adatbázis-kezelő pályája kutatási projektként, mely az üzleti frontra lépés után több tulajdonosváltás után került a mostani tulajdonosához. Érdekesség, hogy több relációs adatbázis-kezelő rendszer is az Ingres-ből nőtte ki magát, így például a mai PostgreSQL is.
[3] A MySQL gyártója eredetileg MySQL AB cég, melyet előbb a Sun Microsystems kebelezett be (2008), majd az Oracle Corp.-hoz került a Sun megvásárlásakor (2010).
[4] Az Oracle adatbázis-kezelő verziószámában az R1 a Release 1, azaz a főverzió 1. kiadás megfelelője.

2. Az Oracle története

A relációs adatbázis-kezelők megjelenésével párhuzamosan, az 1970-es évek elején a CIA egy projektet indított el annak reményében, hogy olyan tudásbázist hozzon létre, amelynek segítségével úgymond „valamennyi kérdésére megkaphatja a választ”. A „kinyilatkoztatás, prófécia” kódnevet adták neki, azaz „Oracle”-re keresztelték el. A projekt ugyan forráshiány miatt abbamaradt, azonban a munkálatokban résztvevő három szakértő: Larry Ellison (a vállalat jelenlegi elnöke), Bob Miner és Ed Oates 1977-ben megalapította a Software Development Laboratories nevű céget, amely 1983 óta viseli a jelenlegi Oracle Corporation nevet. Az alapítás után nem sokkal piacra került az első Oracle nevű adatbázis-kezelő (a CIA volt az első vásárlója a terméknek). Ma az Oracle minden számottevő operációs rendszerre és hardver architektúrára elérhető és telepíthető.

3. Az Oracle felépítése

Az Oracle relációs (valójában ún. objektum-relációs) adatbázis-kezelő rendszer; karbantartására és használatára egyaránt egy szabványos nyelvet, az SQL-t, pontosabban az SQL:2011-et használhatjuk. Az első labor keretében megismerkedünk az Oracle kezelését nagymértékben egyszerűsítő Oracle SQL Developer program menedzsment-szolgáltatásaival. Az SQL lekérdező részeivel (és az SQL Developer ahhoz kapcsolódó felületével) a következő laboron fogunk mélyebben megismerkedni. Mivel a későbbiekben is intenzíven fogjuk használni, ezért a következő laboron az SQL mélyreható ismeretét fogjuk számon kérni.
A teljes Oracle adatbázis-kezelő rendszer fontosabb tulajdonságai a következők:

- Alapvetően kliens-szerver felépítésű.
- Az operációs rendszertől függően lehetővé teszi a többtaszkos, több felhasználós működést, az adatok egyidejű használatát.
- Térben elosztott rendszerként is képes működni.
- A fontosabb hálózati protokollokkal és operációs rendszerekkel együtt tud működni.
- Támogatja a szoftver- és alkalmazásfejlesztés minden egyes szakaszát.
- Képes együttműködni a lényegesebb fordítókkal és fejlesztői környezetekkel.
- Gyakorlatilag tetszőlegesen nagy adatmennyiséget is képes kezelni (különböző hatékonysággal).
- Napi 24 órás rendelkezésre állást, biztonságos működést tud garantálni.
- Magas szinten képes biztosítani az adatok védelmét, integritását, konzisztenciáját.
- Alkalmas összetett struktúrák (objektumok, multimédia adatok, eljárások) tárolására is.
- Fejlett rendszerfelügyelet biztosítható az Oracle Management Server és a hozzá kapcsolódó Agentek segítségével. Ekkor az Enterprise Manager[5] alkalmazás segítségével egy tetszőleges méretű adatbázis-park adminisztrálása/távfelügyelete válik lehetővé.
- Az Oracle, mint cég megbízható terméktámogatási rendszert nyújt a felhasználóinak.

A *szerver* (server) alatt minden esetben egy *adatbázis*t (database) és egy *szerverpéldány*t (instance) értünk. Az adatbázisban tárolódnak a felhasználói és rendszeradatok[6], míg a szerverpéldány a szolgáltatás futtatásához szükséges folyamatok (és szálak[7]) összessége. Egy szerverszámítógép több adatbázisnak is helyet adhat. A legmagasabb szintű, névvel ellátott

[5] Az Enterprise Manager többet is nyújt: az operációs rendszer szintű virtualizációtól kezdve egészen az alkalmazásszerverig egy egész cloud-infrastruktúra szoftveres kezelését képes biztosítani.
[6] A 12c konténer-alapú, ún. multitenant architektúrájában külön adatbázisban, az ún. CDB$ROOT-ban kapnak helyet a rendszer-szintű adatok és metaadatok. A felhasználói adatbázisban csak a felhasználói adatok és az ezekhez kapcsolódó metaadatok foglalnak helyet.
[7] A 12c verziótól kezdődően.

tárolási egység tehát az adatbázis, ami ennek megfelelően jóval több az adatok összességénél! A szerverpéldány a szerver működésének, futásának módját határozza meg. Ilyen például a klaszterezés (Oracle RAC), replikációs beállítások, stb.
A 12c verzió nagy újdonsága volt az ún. multitenant architektúra, ahol a szerverpéldányhoz több[8] felhasználói adatbázis (pluggable database, PDB) is tartozhat. A PDB-ktől különválasztották a rendszerszintű adatokat és metaadatokat egy konténer-specifikus adatbázisba (neve: CDB$ROOT), amelyből konténerenként pontosan 1 példány van. Mivel egy szerverpéldány van jelen, az ahhoz tartozó adatbázisok futásának módja közös, és konténerenként csak egyszer kell a szoftver-karbantartási lépéseket is végrehajtani. A szerver konfigurálható a hagyományos felépítésben történő üzemre is (ez az ún. non-CDB üzemmód), de a felhasználói programok futása szempontjából átlátszó, hogy multitenant vagy hagyományos konfigurációban fut-e az adatbázis-kezelő. A továbbiakban, ha az ellenkezőjét külön nem írjuk, az Oracle Database hagyományos felépítését mutatjuk be, hiszen a labor keretében csak ezt van lehetőség megismerni.
Az adatbázis jól szétválasztható egy fizikai és egy logikai egységre, amelyek külön-külön is karbantarthatóak. A két egységet és a közöttük lévő kapcsolatokat, összefüggéseket az alábbiakban mutatjuk be.

3.1. Logikai felépítés

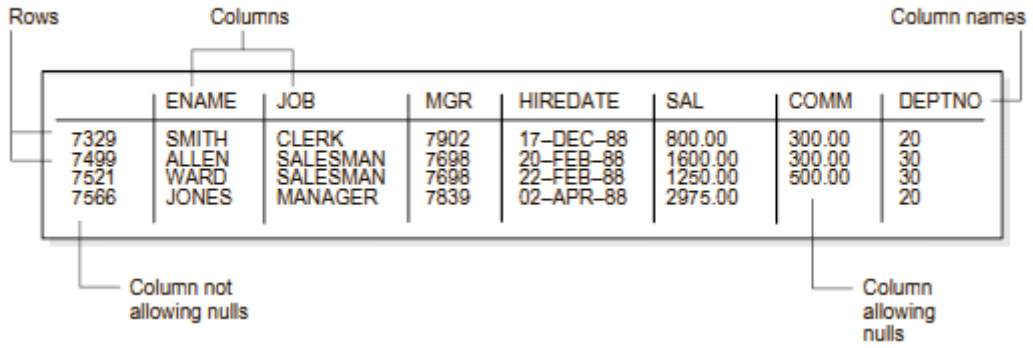
Az adatbázisokat *táblahely*ekre (tablespace, egyes magyar fordításokban táblatér) oszthatjuk fel, amelyek a tárolás logikai egységeit határozzák meg. A táblahely a legnagyobb logikai tárolási egység. A hasonló működésű és karbantartást igénylő adatok kerülnek célszerűen egy táblahelybe. A táblahelyek lehetőséget biztosítanak arra, hogy az adatbázis adatainak elhelyezését kézben tarthassuk; elosszuk különböző tárolási egységek között, kvótákat határozhassunk meg az egyes felhasználók számára stb. Minden esetben van legalább egy táblahely, amelyet **system**nek nevezünk. Ennél természetesen lényegesen több táblahelyet is létre lehet hozni, sőt, általában nem szerencsés felhasználói adatokat a system táblatérbe tenni.
Egy jellegzetes rendszer általában a következőket foglalja magába:
system: a rendszerről tárolt információkat tartalmazó táblahely (vö. adatszótár, data dictionary),
sysaux: kiegészítő táblahely a *system* mellett, amely a 10g verzióban jelent meg. Az Oracle adatbázis néhány olyan funkcionalitása, amelyek korábban a system, vagy különálló táblahelyekben kaptak helyet, most a sysauxot használják. Ilyen például a LogMiner (az adatbázis naplóállományainak programozói feldolgozását biztosító csomag) vagy az Oracle Data Mining opció (adatbányászat csomag).
rbs: az adatbázison végzett műveletek eredményeit, naplóit tartalmazó táblahely az Oracle 9 előtti verziókban (lásd még később a rollback szegmensekről írtakat). Az újabb verziókban hasonló szerepet tölt be az **undo** tablespace.
temp: mindenféle átmenetileg tárolandó adat számára fenntartott táblahely (pl. egyes lekérdezések részeredményei, rendezések eredményei),
tools: alkalmazások ill. általános eszközök által használt minta-táblahely (például a Form Builder is ezt használja),
users: általános felhasználói minta-táblahely.
Az utóbbi két táblahely létezése nem kötelező.
A táblahelyek összességében rendszer- (system) vagy felhasználói (user) objektumokat, egységeket tartalmaznak. Ezek az objektumok lehetnek *táblák* (table), *nézetek* (view), *számlálók* (sequence), *szinonimák* (synonym), *indexek* (index), *csoportok* (group), klaszterek (cluster), *kapcsolódási pontok* (database link) stb., illetve ezekből képzett olyan összetett

[8] Legfeljebb 252 db.

struktúrák, mint amilyen például az *adatszótár* (data dictionary), az (Oracle) *séma* (schema), vagy a futtatható (tárolt) objektumok (stored procedures).
Egy objektum csak egyetlen táblahelynek lehet része, az összetett struktúrák azonban átnyúlhatnak több táblahelyen is.

Fő felhasználói objektumok

Tábla (Table): a logikai adattárolás alapegysége, amely sorokból és oszlopokból áll. Megfeleltethető egy relációnak. A táblát egyértelműen azonosíthatjuk a tábla nevével (pl. SCOTT.EMP). Az oszlopokat a nevük, a típusuk és a méretük jellemzi. Egy táblában csak egyféle típusú adatrekordot tárolhatunk. A sorok sorrendje lényegtelen. Egy tábla létrehozásakor meg kell adni a tábla nevét, valamint a tábla oszlopainak nevét, típusát és méretét. Az alábbi ábra egy táblát, és az abban tárolt adatokat mutatja.
Nézet (View): egy vagy több táblából összeszerkesztett adatok megjelenítésére alkalmas felhasználói objektum. Felfogható úgy is, mint egy tárolt lekérdezés; se nem tartalmaz, se nem tárol (fizikailag) adatot, csak származtatja az adatokat azokból a táblákból, amelyeken értelmezték. Az ilyen táblákat hívjuk a nézet alaptábláinak (base table, master table).
Számláló (Sequence): sorfolytonos, egyedi számgenerátor. Általában valamilyen egyedi azonosító létrehozására használjuk. Fontos, hogy értéke nem tranzakció-orientált, tehát pl. egy rollback művelet nem módosítja a számláló értékét.



1. ábra: Tábla (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 5-2: The EMP Table)

Szinonima (Synonim): egy táblára, nézetre vagy számlálóra több név is megadható a szinonimák segítségével. Lehetőségünk van tehát rövidíteni vagy átlátszóvá tenni az egyes objektumok tárolási helyét. Van nyilvános (public) és rejtett (private) szinonima is. A nyilvános szinonima mindenki számára hozzáférhető, míg a rejtett szinonima csak a felhasználók egy meghatározott körének érhető el. A nyilvános szinonima létrehozása és eldobása speciális jogokhoz köthető.
Index (Index): adatokhoz való hozzáférést (általában) gyorsító eszköz – az Oracle-ben alapesetben egy B* fa. Az esetek többségében olyan táblaoszlop(ok)ra érdemes ilyet létrehozni, amelyre gyakran fogalmazunk meg keresési feltételt. Az indexek automatikusan létrejönnek mindazokra az oszlopokra, amelyekre már a tábla megadásakor megköveteljük az egyediséget. Az indexek frissítését a rendszer automatikusan elvégzi. Az index lehet összetett, azaz több mezőből álló index is, ilyenkor érdemes a nagy kardinalitású oszlopokat az attribútumlista elejére venni. Az indexek létrehozása konkrét esetekben (éles, erőforrás-igényes környezetekben) gondos tervezői munkát igényel, gondatlan megválasztása lassíthatja a működést.

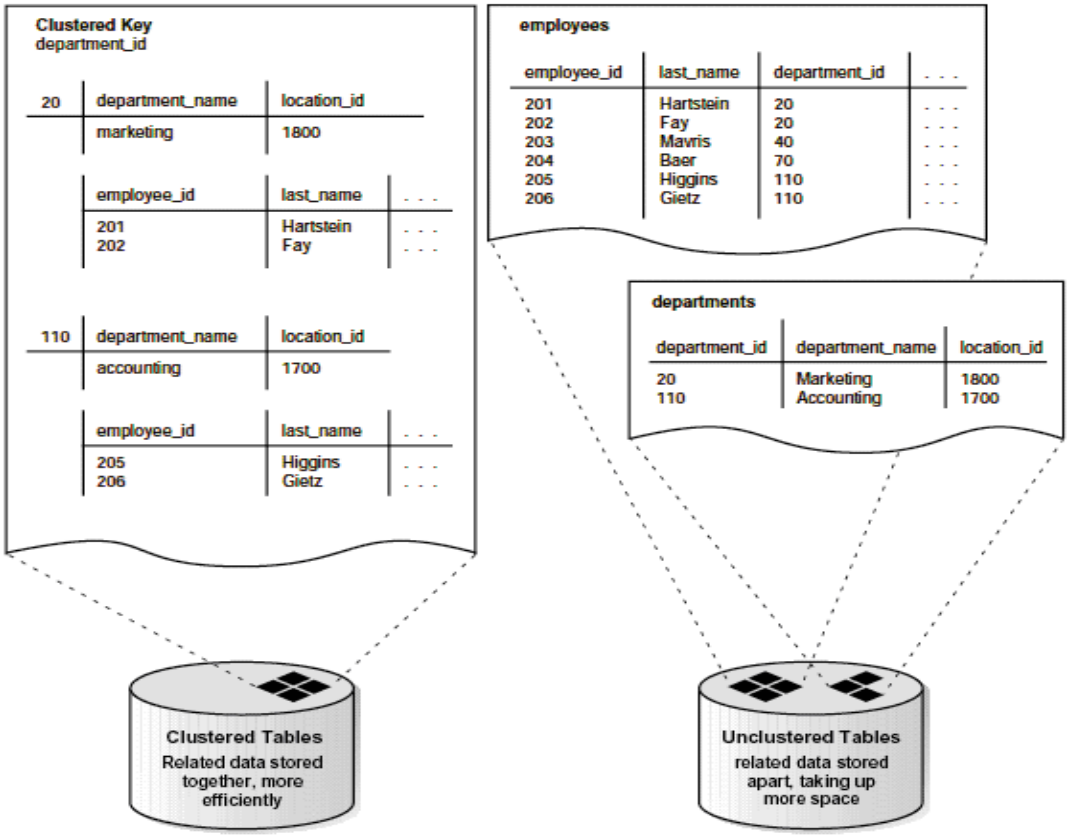
Csoport (Group): Több szervert összefogó struktúra az Oracle 9i-ben. Szerepe a rendszerfelügyelet egyszerűsítése.

Kapcsolódási pont (Database Link): olyan szinonima, amelyen keresztül nem objektumokat, hanem adatbázisokat érhetünk el. Említettük, hogy egy szerverszámítógép több adatbázist is tartalmazhat, sőt lehetnek akár elosztott, azaz több, különböző számítógépen tárolt, azonban adatait tekintve összefüggő adatbázisok is. Ilyen esetekben szükségünk lehet kapcsolódási pontok definiálására.

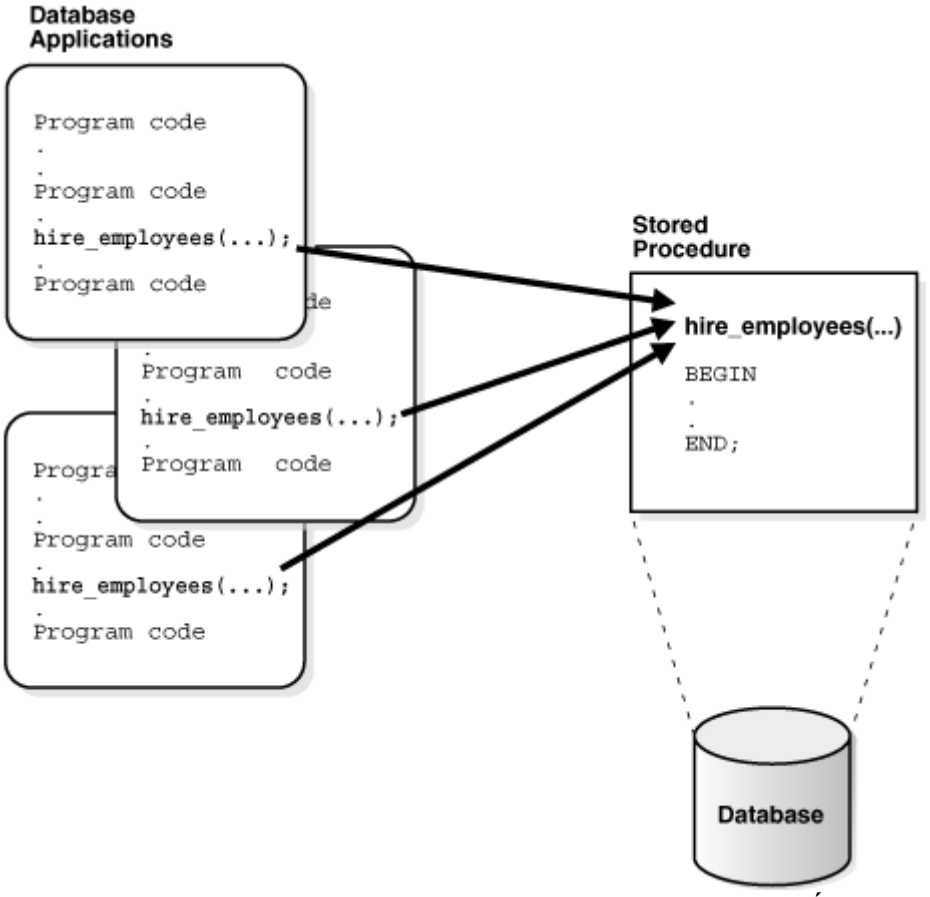
Adatszótár (Data Dictionary): csak olvasható táblák és nézetek gyűjteménye, amelyek a rendszer mindenkori állapotát rögzítik. Ennek megfelelően megtalálható benne, hogy milyen felhasználók vannak a rendszerben, azok mely objektumokhoz férhetnek hozzá; milyen kényszereket kell érvényesíteni az egyes mezőkre; milyen alapértékek vannak beállítva az egyes oszlopokra; mennyi helyet foglalnak az egyes objektumok, mennyi hely van még szabadon; ki, mikor lépett be az adatbázisba és mit módosított vagy nézett meg stb.

Séma (Schema): egy adott felhasználó saját objektumainak összességét nevezzük sémának, vagy a felhasználó sémájának. A felhasználó és sémája között 1-1 értelmű megfeleltetés áll fenn. Az objektumok elérhetőségét a jogosultsági rendszer beállításai korlátozhatják, amelyek a felhasználóra ill. a felhasználók csoportjaira vonatkoznak, ily módon az egyes felhasználók számára biztosítható a sémáikon kívüli objektumok elérése is.

Klaszter (Cluster): az azonos kezelési vagy hozzáférési módot igénylő adatokat érdemes egyetlen csoportba, fizikai helyre tenni. Ha az összetartozó adatok fizikailag „közeli" helyeken vannak, akkor adatbehozatalkor a hozzáférési idő jelentősen csökkenhet. Tehát a csoportokba szervezéssel a hatékonyságot lehet növelni. A 2. ábra egy tipikus példát mutat.



2. ábra: Klaszterezett (bal oldal) és klaszterezés nélküli adatszervezés (jobb oldal) (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 5-10: Clustered Table Data)



3. ábra: A tárolt eljárások az adatbázisban tárolódnak és a szerver futtatja őket. (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 25-2: Stored Procedure)

Tárolt eljárások (Stored Procedures, Functions, Packages): az adatbázisban tárolt, és ott futtatható objektumok összessége. Az adatbázis táblahelyeiben lehetőség van (célszerűen a széles körben használt) szerveren futó, végrehajtható objektumok (pl. PL/SQL, Java, illetve megfelelő beállítások esetén egyéb forrásnyelvi, ún. programok/programrészletek) tárolására is. Az Oracle rendszer installálásakor számos „gyári" tárolt eljárás kerül telepítésre, amelyek pl. megkönnyítik a rendszer adminisztrálását, fejlesztését (3. ábra).
Az egyes objektumok – ahogyan korábban utaltunk rá – elemi adattípusokból épülnek fel. Az Oracle-ben a következő adattípusokkal fogunk találkozni:

Adattípus	Rövid leírás
CHAR (*n*)	Állandó méretű karakterfüzér-típus. Állandó, azaz mindig az előre megadott méretű helyet foglalja le számára a rendszer, függetlenül attól, hogy a karakterfüzér kitölti-e a rendelkezésre álló helyet, vagy sem. Amennyiben a beillesztett szöveg nem tölti ki a teljes méretet, úgy az adatbázis-kezelő kiegészíti a végén a megfelelő számú szóközzel. (Ezt lekérdezéskor is figyelembe kell venni!) A típus maximális mérete az Oracle8-as sorozattól kezdve 2000 bájt. Fontos hangsúlyozni, hogy alapértelmezésben az adatbázis illetve a munkamenet beállításától függ, hogy bájtokban vagy karakterszámban adjuk meg az adatok lehetséges legnagyobb méretét, hiszen van több bájtos karakterből álló karakterkészlet (pl. az UTF8) is. Ezért javasolt explicit megjelölni a hossz-szemantikát a zárójelben a hosszúságot jelző szám után a „char" ill. „byte" szó megadásával. Ha nem

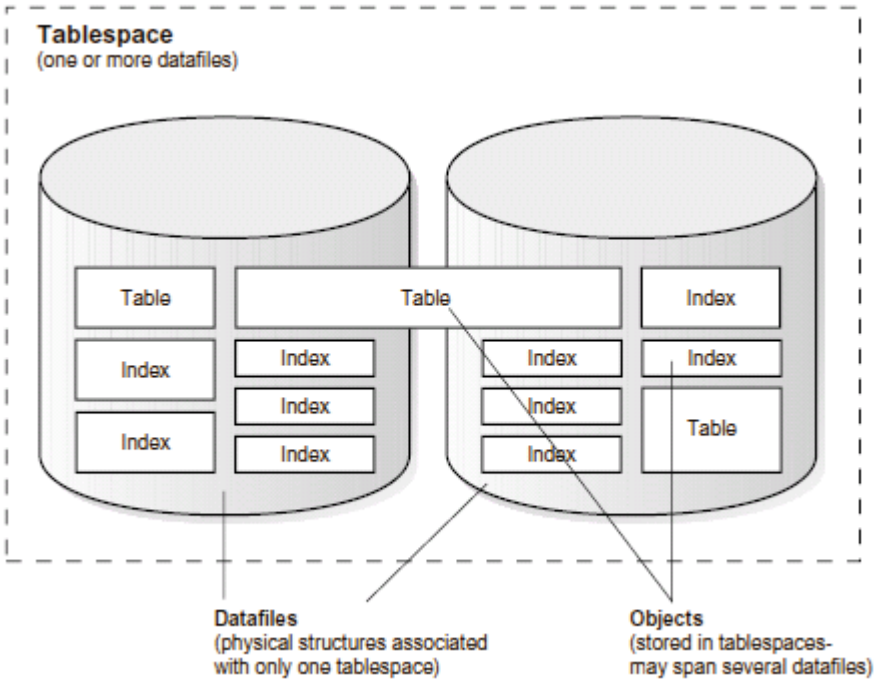
	adjuk meg az adott típusú mező méretét, akkor az alapértelmezés szerint a rendszer azt 1 bájtnak fogja venni. Az adattípusban használt karakterkódolás, és így az ábrázolható karakterek köre az adatbázis karakterkészlet-beállításától (database character set) függ.
VARCHAR2 (*n*)	Változó hosszúságú karakterfüzér-típus. A CHAR típussal ellentétben ennél a típusnál nem egészíti ki a szöveget szóközökkel a maximális hosszra a rendszer, és általában csak a ténylegesen felhasznált méretet foglalja le az Oracle, így használata sokszor indokolt, jobb hatásfokú. A típus maximális mérete az Oracle8-tól 4000 bájt lehet, 12c-től kezdődően 32767 bájt lehet az adatbázis-kezelő megfelelő konfigurációja esetén.
NCHAR (*n*) NVARCHAR2 (*n*)	Az NCHAR és az NVARCHAR2 rendre a CHAR és VARCHAR2 adattípusok Unicode megfelelője, bájtokban vett maximális méretükre ugyanazok a korlátok vonatkoznak. Az ilyen típusú mezőkben Unicode karakterláncok tárolhatók, függetlenül az adatbázis karakterkészlet-beállításától. A típus maximális méretét (n) minden esetben karakterekben kell megadni.
CLOB (LONG)	Nagyméretű szövegek tárolására alkalmas típus. Amennyiben fentieknél nagyobb méretben szeretnénk karakterfüzért tárolni (nem kell megadni felső korlátot), akkor érdemes a megfelelő mezőt CLOB-nak *(Character type Large OBject)* definiálni. A CLOB-nak is van maximális mérete, de ez kellően nagy: elméletileg 4 gibiblokk[9] is lehet. A korábbi Oracle verziókkal való kompatibilitás miatt megmaradt a LONG típus is, ami a megvalósítását tekintve szintén CLOB adatstruktúra. A kettő azonban nem azonos. Lényegesebb különbségeket kiemelve: a LONG típus 2 gibibájtban[10] limitált; egy sémaobjektum csak egy LONG típust tartalmazhat, míg CLOB-ot tekintve korlátlan sokat; a LONG típus csak soros, míg a CLOB véletlen hozzáférésű is lehet.
NUMBER (*p*, *s*)	Tetszőleges szám ábrázolására alkalmas adattípus. Egyaránt használható fix- és lebegőpontos számábrázolásra. Ábrázolási tartománya azon számok halmaza, amelyek abszolút értéke a $[10^{-130}, 10^{126})$ intervallumban van. A számok esetében kétféle méret is megadható; az első a szám tízes számrendszerbeli helyi értékeinek a számát (*p*), míg a második a pontosságot (*s*), azaz a tizedes vessző után álló helyi értékek számát jelenti. Tehát ha egy legfeljebb ±999,99-ig terjedő számot, két tizedes pontossággal szeretnénk ábrázolni, akkor azt NUMBER(5, 2) formában kell megadnunk. Egész számokat a NUMBER(*p*) típussal definiálhatunk, amely ekvivalens a NUMBER(*p*, 0)-val.
DATE	Dátumok megjelenítésére és tárolására alkalmas típus. Az Oracle valamennyi olyan dátumot képes tárolni, amely i.e. 4713. január 1. és i.sz. 9999. december 31. közé esik. A dátum hét darab mezőből áll: század, év, hónap, nap, óra, perc, másodperc. Számos további származtatott egysége is hozzáférhető, úgymint a hét melyik napja, az év hányadik hete stb. Más időszámítási rendszerek (pl. a pravoszláv, a héber, a kínai, a Julián stb.) is elérhetőek az Oracle-ben, sőt az egyes értékek át is válthatóak egymásra. A dátumkezeléssel kapcsolatban lásd még az Oracle beépített dátumfüggvényeit.
ROWID	Az adatrekordok egyedi logikai és fizikai azonosítója. Minden tábla

[9] A gibi prefixum általánosságban 2^{30}=1024×1024×1024-szorost jelent.
[10] 1 gibibájt=1024 mebibájt, míg 1 gigabájt=1000 megabájt. Hasonlóan létezik kibi, mebi…stb. prefixum is.

	rendelkezik egy ROWID nevű segédoszloppal (pseudo column). Az oszlop elemei a tábla sorait egyértelműen azonosítják. Jellemzően hexadecimális kódolásban tartalmazza a sor fizikai elhelyezkedésére és elérésére vonatkozó információkat. Az ilyen kódolású típust nevezzük ROWID típusnak.
UROWID	Az UROWID típus olyan rekordok logikai egyedi azonosítóját tárolja, amelyek fizikai helye más rekordokon végzett műveletektől, vagy az Oracle adatbázis-kezelő hatáskörén kívül eső körülményektől függ. Az ilyen rekordokat tartalmazó táblákban a ROWID nevű segédoszlop UROWID típusú. Az ún. index-szervezésű táblákban (Index-organized Table, IOT) a rekordok az indexek levelében tárolódnak, amelyek új rekordok beszúrásakor/törlésekor, meglevők módosításakor áthelyezésre kerülhetnek más fizikai blokkba. Az index-szervezésű táblák rekordjainak UROWID típusú azonosítója mindaddig változatlan marad, amíg az elsődleges kulcs értéke változatlan. Az Oracle adatbázison kívül tárolt táblák rekordjainak azonosítói szintén UROWID típusúak.

3.2. Fizikai felépítés

Adatállomány (Data file). Az Oracle a táblahelyek adatait egy vagy több adatállományba helyezi el, de egy adatfájl legfeljebb egy táblahelyhez tartozhat – és ennek megfelelően csak egyetlen adatbázishoz (4. ábra). Az állománykezelésnél (lásd Operációs rendszerek, Számítógép architektúrák c. tárgyak) megismert módon, a tárolni kívánt adatok nem feltétlenül azonnal, az utasítás végrehajtásának pillanatában kerülnek be az adatbázisba, jóval hatékonyabb a szakaszos adatkivitel.



4. ábra: A logikai és fizikai tárolás kapcsolata. (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 3-1: Datafiles and Tablespaces)

„Redo-log" állomány (Redo-log file). Egy Oracle adatbázishoz általában kettő vagy több redo-log (pontos fogalmát és szerepét az Adatbázisok c. tárgy részletesen tárgyalja) állomány tartozik. Ezek összességét nevezzük az adatbázis redo-log állományának. Elsődleges feladata, hogy az adatbázison elvégzett műveleteket tárolja egészen az adatkivitel sikeres befejezéséig. Legalább kettő szükséges, így amíg az egyikbe írjuk a redo log buffer tartalmát, addig a másikat további adatbázisfolyamatok (pl. ARC*n*, ld. később) használhatják Mivel az adatbiztonság szempontjából igen kritikus az üzemszerű működésük, így az Oracle támogatja a különböző tárterületekre elosztott redo-log állományok (redo-log group több fájllal) használatát.

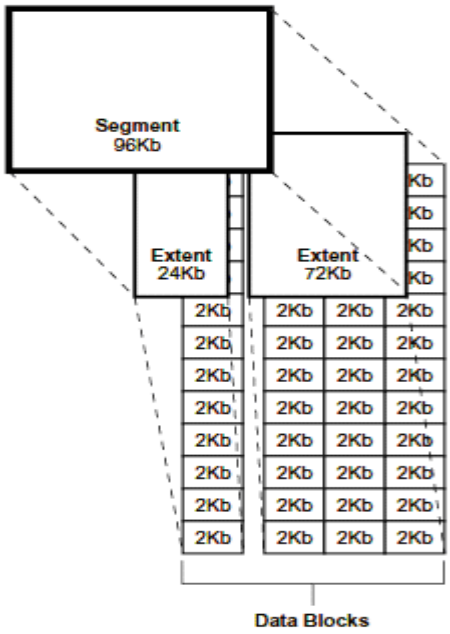
Vezérlési állomány (Control file). Az adatbázis fizikai struktúrájáról tartalmaz információkat. Ilyen információ pl. az adatbázis neve, az adatfájlok neve és fizikai elhelyezkedése, redo-log állományok helye, az adatbázis létrehozásának időpontja stb. (ld. később, az SQL Developer DBA üzemmódjának bemutatásánál). Többnyire egyetlen példány elegendő, de biztonsági okokból ezt is meg lehet többszörözni, el lehet osztani különböző tárterületekre.

3.3. Kapcsolat a logikai és fizikai felépítés között

A fizikai adattárolás logikai egységei három elemből állnak. A legkisebb egység az *adatblokk* (data block, logical block, Oracle block). Az adatblokk általában állandó méretű, összefüggő, az operációs rendszerre vagy magára a (diszken lévő) partícióra jellemző blokkméret (jellemzően 512–4096 bájt) többszörösének megfelelő tárterületet jelent. Az adatblokkok az adatkivitel és az adatbehozatal szempontjából fontosak, hiszen ez az a legkisebb egység, amit az Oracle egy egészként kezel.

Az *extent* (extent) adatblokkok összefüggő halmaza. Az extentek szerepe akkor kerül leginkább előtérbe, amikor egy szegmens – a következő logikai adattároló egység – betelik; ilyenkor az Oracle egy extent méretű hellyel bővíti (egyéb megszorítás hiányában) a használható diszkterületet. Következésképpen egy extent pontosan egy fizikai felépítésre jellemző állományhoz tartozhat.

Ahogy az 5. ábra is mutatja, több logikailag összetartozó extent alkot egy *szegmens*t (segment). Négyféle szegmenst különböztet meg az Oracle, amelyek rendre:



5. ábra: Fizikai adatszervezés az Oracle-ben. (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 2-1: The Relationships Among Segments, Extents, and Data Blocks)

Adatszegmens (data segment): minden táblában megtalálható adat egy ilyenben foglal helyet.
Indexszegmens (index segment): a különféle indexek hatékony tárolására alkalmas szegmens.
Ideiglenes szegmens (temporary segment): minden művelet végrehajtásához az Oracle igényelhet egy ideiglenes munkaterületet, amelyet sikeres befejezés után eldob.
Rollback szegmens (rollback segment): minden megváltoztatott, de még nem committált érték, elem adatát tárolhatjuk itt. Az újabb Oracle verziókban (9-től felfelé) ez a szegmens nem létezik.

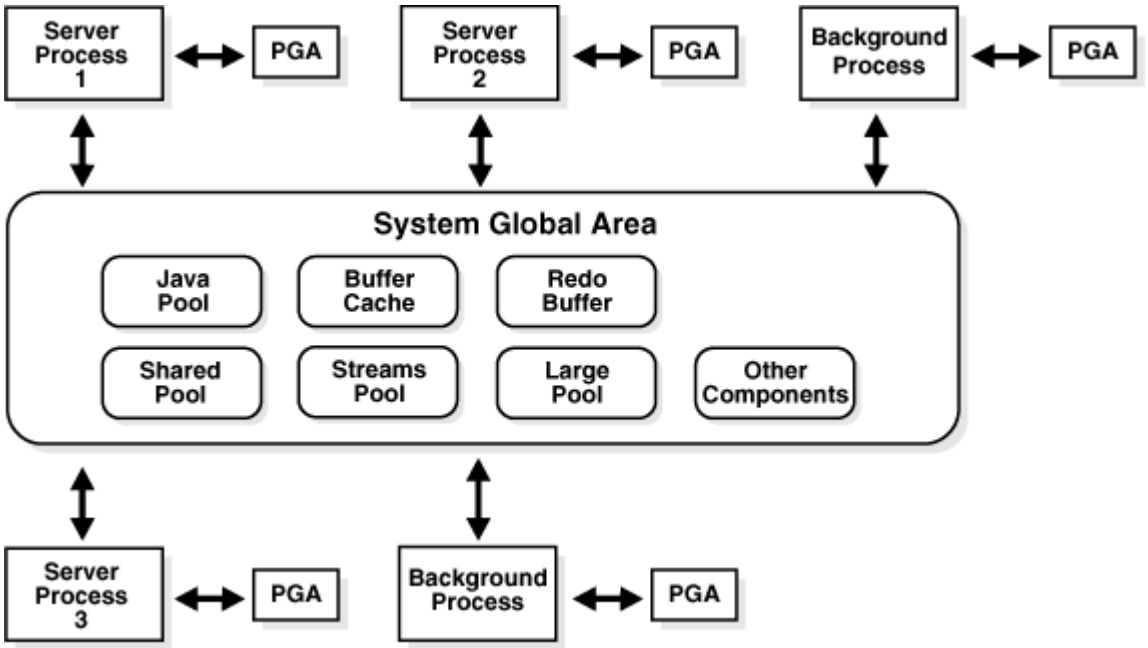
Egy szegmens több adatállományon is átnyúlhat. A szegmenseket a táblahelyek fizikai megvalósításának tekinthetjük.

3.4. A rendszer működése

Az Oracle indításakor a rendszer lefoglal egy memóriaterületet, valamint elindít számos folyamatot (szálat). Ezek együttese alkot egy Oracle példányt. Minden Oracle adatbázishoz tartozik egy Oracle példány, ami az adatbázis üzemszerű működéséért felelős.
Az indításkor lefoglalt *osztott*, a folyamatok számára elérhető memóriaterület az *SGA* (System Global Area). Az SGA mindazon információkat tartalmazza, amelyek az Oracle vezérléséhez szükségesek, másrészt gyorsítótárként is működik: az utolsó használt blokkokat, egy redo-log puffert, az utolsó használt könyvtárak, állományok adatait, az utolsó végrehajtott utasításokat és azok eredményeit (database buffer cache), az Oracle Java folyamatainak memóriastruktúráit, valamint az adatszótárnak egy részletét is magában foglalja (6. ábra).
Az ábrán látható egyes szerverfolyamatokról a későbbiekben részletesen lesz szó.
Az egyes szerverfolyamatok mindegyikéhez lefoglalásra kerül a PGA nevű memóriastruktúra (Program Global Area), amely az adott folyamat állapotát tárolja.

Az Oracle által indított folyamatok részben *rendszerfolyamatok* (server process), részben *háttérfolyamatok* (background process). A rendszerfolyamatok a felhasználót kiszolgáló műveletek: az SQL értelmező és végrehajtó folyamat, az adatkiviteli és -behozatali folyamat a háttértár és az SGA között, valamint a felhasználó számára az eredményeket visszaadó

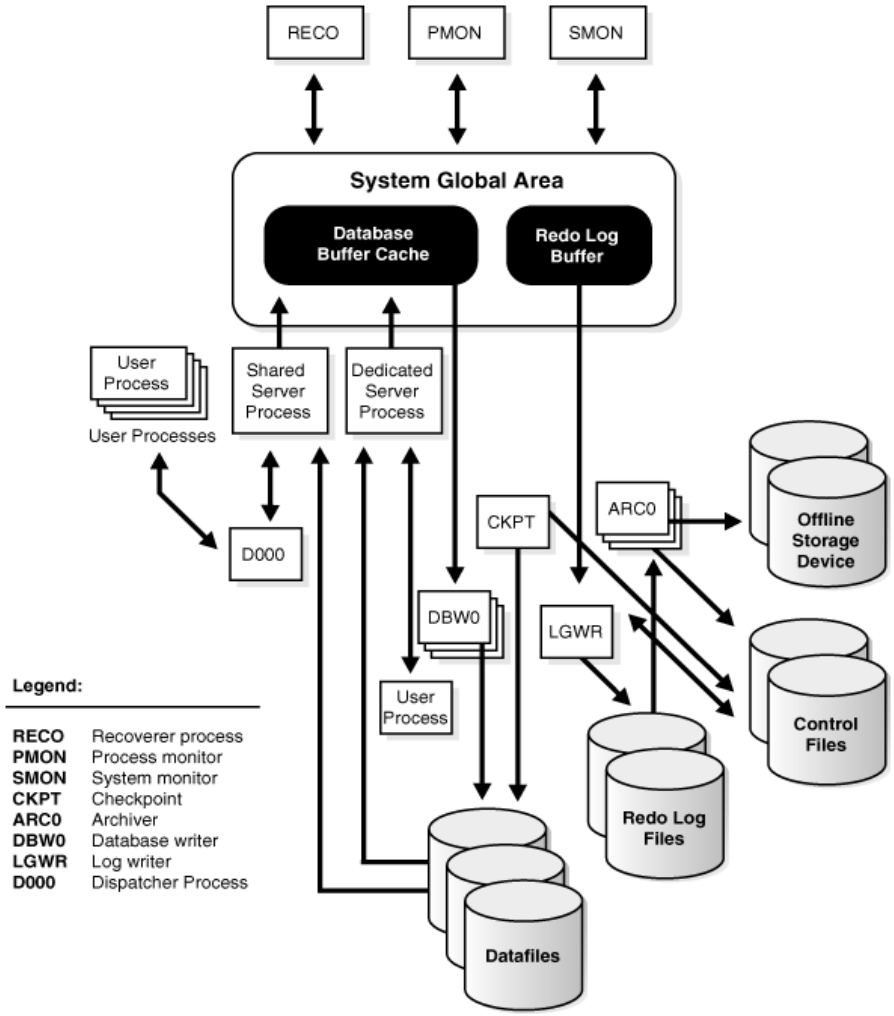
folyamat. A háttérfolyamatok jóval nagyobb számban vannak, a különböző karbantartási feladatokért, a rendszer hatékonyságának megtartásáért felelősek. A legfontosabbak:



6. ábra: A System Global Area (SGA) által tárolt adatok. (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 8-1: Oracle Database Memory Structures)

Rendszerfelügyelő folyamat (system monitor, SMON): a különböző rendszerhibák utáni helyreállítást végző folyamat. Az Oracle indításakor és befejeződésekor automatikusan elindul. Más esetben, szabályos időközönként „felébresztik", hogy megnézze, szükség van-e rá. Ilyenkor az ideiglenes szegmensek már nem használt adatait törli.

Folyamat-felügyelő folyamat (process monitor, PMON): míg az SMON a rendszerhibák után, addig a PMON a felhasználókkal kapcsolatban álló szerverfolyamatok hibái után „takarít". Ha egy ilyen folyamat nem hajtódik teljesen végre, akkor a PMON a felhasználó megfelelő tranzakcióit, zárait és egyéb foglalt erőforrásait felszabadítja.
Adatbázis író folyamatok (database writers, DBW*n*): a szükséges, módosított adatokat írja ki az SGA-ból a háttértárra, a megfelelő adatfájlokba. Legfeljebb 20 ilyen folyamat működhet egyszerre.
Naplókészítő folyamat (log writer, LGWR): a redo-log puffert írja stabil tárba az SGA-ból. Az elvégzett műveleteket az aktív redo-log állományba jegyzi le.
Archívumot készítő folyamat (archiver, ARC*n*): az aktív, betelt redo-log állományt egy erre a célra kijelölt tárra másolja. A másolat célja biztosítani a rendszerhibák utáni helyreállítást. Legfeljebb 10 ilyen folyamat működhet egyszerre, szükség szerint a LGWR gondoskodik új ARC*n* folyamatok indításáról.
Zárfolyamatok (lock manager server processes, LMS): Oracle példányok közötti erőforrás-kezelést valósít meg az Oracle Real Application Cluster-ben (kb. Oracle parallel szerver).
Párhuzamosító folyamat (dispatcher, D000): a párhuzamosító feladata, hogy a felhasználói folyamatok között megossza a kisszámú rendszerfolyamatot (többszálúként beállított szerverek esetében *(shared server üzemmód)*). Így ugyanannyi rendszerfolyamattal jóval több felhasználó szolgálható ki. Jellemzően ez a kapcsolódási pont az Oracle szerver és a (kliensen futó) felhasználói folyamatok között. A párhuzamosítóhoz minden esetben SQL*Net (Net8) protokollon keresztül kell csatlakozni.



7. ábra: Az Oracle háttérfolyamatai. (Ábra forrása: Oracle 11gR1 Database Concepts, Fig 9-2: Background Processes of a Multiple-Process Oracle Database Instance)

Az egyes folyamatok közötti kapcsolatokat mutatja a 7. ábra.
Összefoglalva: Egy Oracle példány a rendszer és háttérfolyamatokból, illetve a rendszer által lefoglalt memóriaterületekből (pl. SGA) áll.

Az Oracle és a felhasználói folyamatok (kliensprogramok) mindig Net8 protokollon keresztül kommunikálnak. A Net8 protokoll elfedi a különböző lehetséges hálózatokat és programozói felületeket (viszony, és megjelenítési szintű protokoll). Így a Net8 illeszthető pl. IPX, SPX, IPv4, IPv6, TCP, TCPS hálózatokra egyaránt. Ugyanezen protokoll felelős a nyelvi beállításokért. Azaz: kizárólag a Net8 protokoll *kliensoldali* (felhasználói) beállításaitól függ, hogy az Oracle milyen nyelven írja ki az üzeneteit, a dátumokhoz tartozó mezőneveket, ill. milyen karakterkészlet szerint rendez.

3.5. Az Oracle biztonsága

Az Oracle rendszerhez az Oracle-ben definiált felhasználók csatlakozhatnak. Ezek a felhasználók és a hozzájuk kapcsolódó jogosultságok (első közelítésben) függetlenek az operációs rendszer felhasználóitól, illetve jogosultsági rendszerétől.
Az Oracle jogosultsági rendszere kétlépcsős. Az első csoportba tartoznak a *rendszerjogosultságok* (*system privileges*, pl. a CREATE TABLE a tábla létrehozására, ALTER SESSION a kapcsolat módjának megváltoztatására), amelyekkel a rendszer

egészével kapcsolatos jogokat állíthatjuk be egy-egy felhasználó számára, továbbá a *felhasználói objektumokra vonatkozó jogosultságok* (*object privileges*).
Az egyes rendszer- és objektumjogosultságokból összeállítható egy úgynevezett szerep (role). A rendszer tartalmaz néhány előre definiált szerepet; ilyen a DBA (adatbázis rendszergazda, DataBase Administrator), a CONNECT[11] (csatlakozási jog az adatbázis-kezelőhöz) és a további jogosultságokat biztosító RESOURCE „jog" is.
Mivel sokszor kényelmetlen lenne az összes szükséges jogosítványt felsorolni egy-egy feladat elvégzésére, így célszerűbb a jogosítványok mindegyikét tartalmazó szerepet megjelölni. Például ahhoz, hogy az adatbázis lemezterületeit karbantarthassuk, mintegy húszféle rendszerszintű jogosultságra van szükség, azonban a felsorolásuk helyett azt mondjuk, hogy DBA jogokkal kell rendelkezni. A felhasználói adatok és az egyes jogosultságok karbantartásához viszont a CREATE/ALTER/DROP USER jogok is elegendőek.

4. Az Oracle üzemeltetése

Az Oracle a távoli adminisztrációt (is) támogatja különféle eszközök segítségével. Az adatbázis (távoli) karbantartásának központi eleme az Enterprise Manager Grid Control, amely webes felületen az adatbázis-kezelő karbantartásán túl számos infrastrukturális elem (pl. Oracle Virtual Machine Server) felügyeletére is tartalmaz eszközöket. A labor keretében az Enterprise Manager helyett az Oracle SQL Developerbe épített DBA üzemmódot fogjuk megismerni.

4.1. Az SQL Developer indítása

Az SQL Developer telepítésére nincs szükség, a letöltött .zip fájl kibontása után az sqldeveloper.exe vagy sqldeveloper.sh fájl futtatásával indítható.
Amennyiben az alkalmazást először indítjuk el, szükség lehet a Java környezet[12] útvonalának megadására.
Az SQL Developerben három üzemmód-választó fül egyes elemeivel fogunk foglalkozni. A három fül a Connections (definiált adatbázis-kapcsolatok), Reports (az adatainkból vagy az adatbázisról készült különböző kimutatások, „riportok") és a DBA (egyes adatbázis-adminisztrátori funkciók).
Az adatbázis-kezelőhöz történő csatlakozáshoz a Connections fülön levő zöld + (plusz) ikonra kattintva meg kell adni annak a szervernek az adatait, amelyhez kapcsolódni kívánunk, az alábbiak szerint:

- *Connection name:* tetszőleges név a kapcsolat azonosítására, pl. szglab5
- *Username/Password:* az adatbázis-kezelőhöz történő felhasználói név/jelszó páros. Nem kötelező kitölteni.
- *Save Password*: a jelölőnégyzetet megjelölve a felhasználónév/jelszó párost elmenthetjük (ha a fenti mezőkben megadtuk), így a csatlakozásnál megadásuk már nem lesz szükséges (a HSZK-ban ne jelöljük be a négyzetet).
- Az Oracle fület választva egy Oracle adatbázis-kapcsolat leírását készíthetjük el az alábbiak szerint:
    - *Connection type*: milyen módon csatlakozzunk az adatbázis-kezelőhöz. A Basic üzemmódot választva egyszerűen megadható a szerver címe, portja, és az adatbázis neve (l. a következő vázlatpontokat). Használhatjuk még a TNS üzemmódot, ekkor egy ún. TNS-leíró állományra[13] van szükség, és az abban

[11] A CONNECT szerep a 10gR1 verzióval bezárólag tipikus felhasználói jogosítványgyűjtemény, amely az adatbázis-kezelőhöz való csatlakozást, néhány típusú objektum létrehozását, használatát teszi lehetővé
[12] Java Development Kit (JDK) szükséges, Java Runtime Environment (JRE) nem elég.
[13] Egy tnsnames.ora fájl, ami a TNS_ADMIN környezeti változó által mutatott könyvtárban (ez az általános megoldás), vagy az SQL Developer Preferences/Database/Advanced/Tnsnames directory helyen van.

definiált kapcsolatleírók közül választhatunk. Advanced üzemmódban pedig tetszőleges JDBC URL megadására nyílik lehetőség, amelynek érvényessége a felhasználó felelőssége.
    - *Role*: meghatározhatjuk, hogy a szervert milyen jogosultságokkal kívánjuk elérni. A mérés során sem a SYSDBA (adatbázis adminisztrátor) sem pedig a (valamivel korlátozottabb jogokat biztosító) SYSOPER szerepet nem fogjuk használni, így a csatlakozás csak „default” (néhány kliensben: „Normal”) módban lehetséges.
    - *Hostname*: a szerver DNS-neve, pl. rapid.eik.bme.hu
    - *Port*: a szerver port-száma. (Általában 1521 vagy, elsősorban Magyarországon, 1526).
    - *SID*: a szerver rendszerazonosítója (System Identifier). Az a név, ahogy az adatbázist az adatbázis-adminisztrátor elnevezi, ez jelen esetben szglab. (Általában nem célszerű 6 karakternél hosszabbra választani.)
    - *Service Name*: az adatbázist, mint szolgáltatást azonosítja.
- Amennyiben minden adatot megadtunk, és mentettük a kapcsolatleírót, az meg fog jelenni a képernyő bal oldalán látható fehér területen. Az adatbázis ikonjára kattintva csatlakozhatunk a rendszerhez, felhasználónevünk és jelszavunk begépelése után. (Ezek az adatok az első mérésen kiosztásra kerülnek.)

Amennyiben minden adatot sikeresen adtunk meg, az adatbázis-szervert reprezentáló ikon mellett egy villásdugó jelenik meg, jelezve a sikeres csatlakozást, egyben az ikon melletti „+” jelre kattintva láthatóvá válik az adatbázist objektumait reprezentáló fa-gráf. A munkamenet végeztével kijelentkezni a kapcsolat nevére történő jobb-klikk után a „Disconnect” paranccsal lehetséges (és ajánlott).
A Connections fül mellett található a DBA üzemmódot megtestesítő fül (ha nem látszik, akkor a View menü DBA pontjával hívható elő). Itt a zöld + (plusz) ikonra kattintva engedélyezhetők a kliens DBA üzemmódjához az egyes, már korábban definiált kapcsolatok. A kapcsolat neve melletti „+” jelre kattintva megjelennek a DBA módban elérhető információk és funkciók egy fa-gráf formájában.
A Reports fül az előző kettővel szemben másképp működik: előbb a futtatni kívánt riportot kell kikeresni a megjelenő fa-gráfban, és azután kell megadni, hogy melyik adatbázis-kapcsolaton és (szükség szerint) milyen paraméterekkel fusson.

A továbbiakban a fent említett fa-gráfok egyes elemeit vizsgáljuk meg részletesebben.

4.2. A szerverpéldány beállításai (DBA üzemmód, „Database Configuration”)

Ezen a ponton érhetőek el és változtathatók meg a rendszer működését alapvetően befolyásoló beállítások, az inicializációs paraméterek. A paraméterek megváltoztatása statikus paraméterek esetén a szerverpéldány újraindítását igényli, míg dinamikus paraméterek esetén erre nincs szükség.
Amennyiben egy adatbázison SYSDBA jogosultságunk van (tehát a mi felelősségünk e paraméterek szabályozása) nem célszerű az adatbázis újraindítást erről a felületről elvégezni. (Az adatbázis újraindítása a Connections üzemmódban a kapcsolat nevére történő jobb klikk után a Manage Database pont alatt lehetséges.)

4.3. Munkamenetek/Sessions (Reports fül)

A Reports fülön a „Data Dictionary Reports/Database Administration/Sessions” alatt érhetők el különböző szempontok szerint a munkamenetek adatai: erre a pontra lépve az adatbázissal aktuálisan kapcsolatban lévő, szerveroldalon futó folyamatok megtekintésére és szabályozására nyílik mód (pl. egy megakadt, vagy káros folyamat „kilövésére” a Sessions

nevű riportban). Az itt rendelkezésre álló lehetőségek közül a legérdekesebb talán az SQL analízis, ahol egy futó folyamat által kiadott SQL parancsot lehet megtekinteni. Ez a lehetőség a gyakorlatban használható pl. szoftverfejlesztésnél egy kritikus SQL utasítás vagy tárolt program felderítésére és felgyorsítására, a feldolgozási lépések jobb megértésén keresztül.

4.4. Zárak/Locks (Reports fül)

A munkamenetekhez hasonlóan a Reports fülön, a „Data Dictionary Reports/Database Administration/Locks" pont kiválasztásával tekinthetjük meg a rendszerben jelenleg aktív zárakat. (Azokat a zárakat is, amelyeket pl. az SQL Developer illetve az Oracle belső folyamatai helyeztek el.) A gyakorlatban ez a képernyő a szoftverhibák megkeresését segítheti elő.
Az itt megjelenített információ a V$LOCK nézeten keresztül kérdezhetőek le SQL felületről.

4.5. Az adatbázis tartalmának kezelése („Schema Manager")

A Connections fülön, az adatbázis-kapcsolat neve alatt találhatóak a felhasználó saját objektumai típus szerinti bontásban (az „Other Users" alpontban a többi felhasználó objektumai érhetőek el).
Itt nemcsak objektumok egyszerű (varázslók segítségével történő) létrehozására van lehetőség, hanem egyrészről az objektumokhoz kapcsolódó speciális beállítások végezhetők el (pl. constraintek felvétele, tárolási jellemzők beállítása, triggerek és programok definiálása, amelyet pl. syntax highlighting segít), másrészről pedig az objektumok tartalmának grafikus felületről történő módosítása is lehetséges (pl. egy tábla kitöltése, vagy egy nézet gyors felvétele).
Az Oracle számos, a sémákban elhelyezett objektumok kezelését megkönnyítő eszközt bocsát rendelkezésünkre, általában beépített nézetek formájában. Így pl. az adatbázis-adminisztrátor kikeresheti az összes olyan objektumot, amelynek nevére egy meghatározott karakterlánc illeszkedik.

4.6. Alapvető biztonsági beállítások (DBA üzemmód, Security)

Az adatbázishoz és annak adataihoz való hozzáférés-szabályozás elemei tekinthetőek meg és módosíthatók itt. A megfelelő jogosultságok általában felhasználóhoz kötöttek, így szükséges a felhasználó azonosítása. Erre a célra a méréseken a „hagyományos" felhasználónév-jelszó páros szolgál, de az Oracle lehetőséget biztosít erős titkosításon, illetve nyilvános kulcsú titkosítást használó szoftverarchitektúrán (PKI) keresztül történő azonosításra is (ld. az Oracle Wallet, illetve az Enterprise Security Manager alkalmazást az ún. OCI kliensben). Felhívjuk a figyelmet arra, hogy az Oracle rendszer alapértelmezésben nem használ titkosítást az adatátvitel során (kivéve a jelszavakat), ennek beállítása az adatbázis-adminisztrátor feladata. (Ajánlott az egyszerű SSL alapú titkosítás. Ennek hátránya, hogy megnöveli a kommunikációhoz szükséges sávszélességet, illetve a bejelentkezés időtartamát.)
Elosztott adatbázisrendszerek esetén lehetőség van ehhez idomuló bejelentkezési rendszer megvalósítására (ld. Enterprise Logon Assistant).
A User alpont jobb-klikk Create new parancsával új felhasználót is itt lehet a rendszerhez adni.
A Security ponthoz kapcsolódó fontosabb párbeszédpanel-fülek az alábbiak (nem mindenhol van jelen mindegyik):

- *User*: Itt nyílik lehetőség a felhasználói név és jelszó beállítására, a felhasználó alapértelmezett táblahelyének beállítására, a korábbiakban említett ideiglenes táblahelyek beállítására, illetve a felhasználó engedélyezésére/tiltására.
- *Granted Roles*: a felhasználó szerepeit állíthatjuk be.
- *System priviliges*: rendszerszintű privilégiumok szabályozása.

- *Object priviliges*: objektumszintű privilégiumok szabályozása.
- *Quotas*: a UNIX rendszerekhez hasonló kvóták beállítása táblahelyenként.
- *Proxy users*: Ez a lehetőség arra szolgál, hogy az egyes speciális feladatok az adatbázist adott felhasználóként el tudják érni.

4.7. Fizikai tárolási paraméterek (DBA üzemmód, Storage)

Az adatbázis fizikai megvalósításának elemeit lehet vele megtekinteni és karbantartani. Alkalmas egyfelől különböző táblahelyek, adatállományok és rollback szegmensek létrehozására, módosítására és szükség esetén ezek törlésére, másfelől a kihasználtságról, a szükséges hely- és tárigényekről kaphatunk részletes információkat. Fontos eleme az ún. High Watermark, amely jelzi az adott objektum létrehozása óta annak valaha előfordult maximális kihasználtságát. Változtatások végrehajtásához DBA jogosultsággal kell rendelkezni.

5. Függelék

A függelék nem tartozik szorosan a mérés anyagához, nem fogjuk számon kérni a tartalmát.
A laborok során nem fogjuk kihasználni az egyes Oracle verziók speciális tulajdonságait. Ahol ezt meg kellett tennünk, ott felhívtuk a figyelmet a változásokra. Mindazonáltal fontosnak tartottuk, hogy a hallgatók érzékeljék a különbségeket, a fejlődési trendeket.

5.1. Az Oracle újabb verzióinak összehasonlítása

Oracle 8

Az Oracle8 az első objektumrelációs adatbázis-kezelő az Oracle sorozatban, amely 1997-ben jelent meg. Leegyszerűsítve ez annyit jelent, hogy az Oracle képes objektumok adatainak és eljárásainak tárolására, megjelent a típus fogalma és lehetőség nyílt a multimédia adatok hatékonyabb kezelésére.
Az adattípusokat illetően a legjelentősebb változás, hogy a LONG helyett a LOB típusokat lehet már használni, ami sokkal dinamikusabb és hatékonyabb. A CHAR és VARCHAR2 maximális mérete is megváltozott, a korábbi 255 ill. 2000 helyett 2000 ill. 4000 lett.
Az alkalmazásfejlesztésekhez kibővítették a JDBC, azaz a Java nyelven keresztüli adathozzáférés képességeit. Az adatbázisból lehetőség van adatbázison kívüli függvényhívásokra is akár HTTP vagy IIOP (egy CORBA szabvány) protokollokon keresztül.

Oracle 8i

Az Oracle-t kifejezetten Internetes alkalmazások támogatására alakították át, megjelenése 1999. A telepítő és a kliensek lényegében Java-alapúak, illetve a beépített Java VM segítségével a szerver maga is képessé vált Java alkalmazások futtatására. Különálló termékként megjelent a WebDB, ami a korábbi Webservert váltotta fel egy sokkal hatékonyabbra. A Webserver PL/SQL segítségével állította elő a HTML oldalak tartalmát. A WebDB alkalmazásban a két dolog felcserélődött, varázslók segítségével rakhatjuk össze a HTML oldalt, a PL/SQL forrás automatikusan generálódik.
A szerver magában foglalja InterMedia csomagot is, amivel multimédia adatok internetes megjelenítését, tárolását, lejátszását és továbbadását is támogatja, de arra is lehetőséget biztosít, hogy különböző lekérdezéseket, riportokat kérjünk le HTML, PDF, Word vagy Excel formában. Mindezek az EDI (Electronic Document Interchange) támogatását szolgálják.
Jelentősen kibővítették a DATE adattípushoz tartozó elemeket, elsősorban a konverziók terén. Például belekerült a fél hónap, 10 nap és a félév fogalma is. De a dátumokat szabadon lehet felüldefiniálni, például az üzleti világ elvárásaink és követelményeinek megfelelően. Az SQL utasítások között megjelent a DROP COLUMN utasítás, azaz lehetőség nyílt oszlopok törlésére. A ROWID típust kiterjesztették UROWID típussá, amely félig logikai kulcs, hiszen a táblák elsődleges kulcsainak kitöltésétől is függ – ezáltal gyorsítva az adathozzáférést.

Oracle 9i

Az adatbázis-kezelő jelentősebb belső átalakításon ment keresztül, amely leginkább az adatbányászat és az adattárházak területeit érintik. A 9i Release1 2001-ben jelent meg. A legfontosabb változás, hogy az Oracle9i SQL-99 kompatibilis lett (korábban csak SQL-92 kompatibilitást biztosítottak).
A korábbi Oracle termékekkel ellentétben, ebben a verzióban már lehetőség van az SGA területének és tartalmának dinamikus, azaz futási időben elvégezhető módosítására – ezek is bekerültek a megfelelő SQL utasítások közé. Ez változat már a különböző blokkméretek kezelésére is lehetőséget biztosít, sőt, akár külső adatbázisbeli felhasználói objektumok elérésére is nyújt interfészt.

Oracle 10g

Az Oracle 10g első kiadása (R1) 2003-ban, a Release 2 2005-ben jelent meg, a nevében a „g” a „Grid computing” kifejezésből származik. A nevével összhangban az elosztott erőforrások és szolgáltatások egy logikai egységként történő kezelésében tartalmaz számos előrelépést, valamint diagnosztikai és tuning-fejlesztéseket. Az elosztott működés nem a 10g újdonsága: az Oracle adatbázis-kezelő már korábban is tartalmazta a Real Application Clusters[14] technológiát.
A 10g verzió számos további újdonsága a logikai és fizikai tárolási rétegek köré csoportosul. A teljesség igénye nélkül megemlítünk néhányat közülük. Az automatikus tárkezelés (Automatic Storage Management, ASM) egy logikai kötetkezelési réteg, amely az adatbázis-kezelő alatti platform tárkezelő mechanizmusaitól független kötetkezelést tesz lehetővé. Míg az Oracle korábbi verzióiban táblahelyet csak azonos platformon futó adatbázisok között lehetett másolni, a 10g-től kezdődően erre különböző platformok esetén is van lehetőség.
Az újonnan megjelent lomtár (recycle bin) a törölt adatbázis-okjektumok tárolási helye (amennyiben engedélyezett a szerverpéldány szintjén), ahonnan azok szükség szerint visszaállíthatók. Az objektumok szintje melett lehetőség van az adatok szintjén is a visszaállításra az ún. flashback technológiával, amely a 10g-ben SQL utasítások szintjére került (korábban egy PL/SQL csomag volt), és lehetőséget biztosít adatbázis és tábla szinten is a visszaállításra, illetőleg a korábbi állapot lekérdezésére.
A rendszer teljesítményanalíziséhez az AWR (Automatic Workload Repository, automatikus terhelés-repozitórium) rendszeres időközönként feljegyzi a fontosabb teljesítmény-paramétereket, amelyet az ADDM (Automatic Database Diagnostics Monitor, automatikus adatbázis-diagnosztikai monitor) komponens dolgoz fel és tesz elérhetővé.
Egy rendszer fejlődése során időnként elkerülhetetlen, hogy a megjelenő új komponensek mellett korábbiak tűnjenek el vagy változzanak meg. Az Oracle 10g lekérdezés-optimalizálója hivatalosan már nem támogatja a szabály-alapú optimalizálást, és ennek megfelelően automatikusan gyűjti és frissíti a költség-alapú optimalizálót segítő objektumstatisztikai adatokat.

Oracle 11g

2007-ben jelent meg az Oracle 11gR1, a Release 2 pedig 2009-ben. Az R1 kiadás a SQL:2003, míg az R2 kiadás az SQL:2008 szabványok kötelező ún. Core részével nagyrészt kompatibilis. Számos apró újítása a hatékonyabb erőforrás-kihasználást célozza meg mind teljesítmény (pl. statisztika-gyűjtés, lekérdezéseredmény cache), mind tárhely (pl. tömörítés az egyedi DML műveletek eredményében is), mind DBA-erőforrások (pl. automatikus memória-tuning, terhelés-profilok rögzítése és visszajátszása: Real Application Testing) tekintetében.
A táblák a 11g-től kezdődően tartalmazhatnak ún. virtuális oszlopokat, amely a nézetekhez hasonlóan teszik lehetővé SQL kifejezésekkel definiált oszlopok megadását a rekord többi mezőjének értéke alapján. Ez a virtuális oszlop a tábla „teljes jogú” oszlopa lekérdezésekkor és indexek építésekor, ill. a tábla ún. particionálásakor.

Oracle 12c

2013-ban jelent meg az Oracle 12cR1, amely már nagyrészben SQL:2011-es szabvány-kompatibilitást nyújt annak kötelező, ún. Core részével. A verziószámban megjelenő „c” a felhő-alapú számítástechnikára utal (cloud-computing). Ehhez kapcsolódó talán

[14] Oracle Real Application Clusters (RAC), a 9i előtti verziókban Oracle Parallel Server (OPS). Elosztott tranzakciófeldolgozási képességeket megvalósító technológia, amelyben több szerverpéldány (instance) kezeli a közös háttértáron elhelyezett adatbázist. A technológia jól skálázható, magas rendelkezésre állású logikai adatbázis-szervert biztosít.

leglényegesebb újdonság az ún. multitenant architektúra, amely egy konténer adatbázisszerverből, és benne felhasználói adatbázisokból (pluggable database, PDB) áll. Ilymódon a szerverpéldány (instance) közös az egy konténerhez tartozó adatbázisok között. Ez az adatbázisok menedzselését könnyíti meg, hiszen az olyan szerverpéldány-szintű beállítások, mint a replikáció, mentés-csoportok, Real Application Cluster (RAC) az összes adatbázisra érvényesek lesznek, és a DBMS szoftverfrissítéseit is csak egyszer kell telepíteni. Az egyes PDB-k könnyen átcsatolhatók a különböző konténerek között. Szintén a 12c újítása, hogy egyes szerverfolyamatok immár többszálú működésre is beállíthatók.
Az architekturális változtatáson túl a fejlesztések egy jelentős hányada azt célozza, hogy minél egyszerűbb legyen más adatbázis-kezelő rendszerekről Oracle Database-re „portolni" az alkalmazásokat. Ilyen SQL-fejlesztések pl. a top-N lekérdezések megfogalmazására szolgáló szintaxis-kiegészítés a select utasításban, vagy a számláló-jellegű, automatikus kitöltésű azonosító mező generálását kényelmesebbé tevő oszlop-beállítás (identity_clause illetve <*sequence*>.nextval, mint alapérték). Egy egzotikusabb példa ugyanebből a körből, hogy a MySQL C nyelvű API-val kompatibilis felületen keresztül közvetlenül elérhető az Oracle Database adatbázis a liboramysql meghajtó segítségével.
További fejlesztések között említjük a select utasítás idősor-jellegű mintaillesztési képességgel történő felruházását (row_pattern_clause), vagy az adaptív lekérdezési tervek készítését. Ennek lényege, hogy a statisztikák alapján kiválasztott illesztési algoritmust bizonyos feltételek mellett futásidőben megváltoztathatja a DBMS, ha a végrehajtás során, a valódi adatok egy részét feldolgozva úgy találja: jobbat is választhat.

II. labor: Az SQL nyelv

Szerzők: Kiss István anyagát kiegészítette Gajdos Sándor, Unghváry Ferenc

1. Az SQL története

- 1974–75-ben kezdték a kifejlesztését az IBM-nél, az „eredeti” neve SEQUEL (Structured English QUEry Language);
- 1979-től több cég (pl. IBM, ORACLE Corp.) kereskedelmi forgalomban kapható termékeiben;
- 1987-től ANSI szabvány.

2. A nyelv jelentősége

- Szabvány, amelyet jelenleg csaknem minden relációs adatbázis-kezelő alkalmaz (kisebb-nagyobb módosításokkal);
- tömör, felhasználó közeli nyelv, alkalmas hálózatokon adatbázis-kezelő szerver és kliensek közötti kommunikációra;
- nem procedurális programozási nyelv (legalábbis a lekérdezéseknél).

3. A nyelv definíciója

A leírás az ORACLE adatbázis-kezelő SQL dialektusát ismerteti, ez többé-kevésbé megfelel az egyéb termékekben található nyelv variációknak. A nyelv termék- illetve hardverspecifikus elemeit nem, vagy csak futólag ismertetjük.

A nyelv utasításait a következő csoportokra oszthatjuk:

- adatleíró (DDS – Data Definition Statement)
- adatmódosító (DMS – Data Manipulation Statements)
- lekérdező (Queries)
- adatelérést vezérlő (DCS – Data Control Statements)

A nyelvben – a szöveg literálok kivételével – a kis- és nagybetűket nem különböztetjük meg. A megadott példáknál a könnyebb érthetőség miatt a nyelv alapszavait csupa nagybetűvel, míg a programozó saját neveit kisbetűvel írjuk.
A parancsok több sorba is átnyúlhatnak, a sorokra tördelésnek nincs szemantikai jelentősége. Az SQL parancsokat pontosvessző zárja le.

4. Táblák létrehozása, törlése

4.1. Táblák létrehozása

Új táblát a

```
CREATE TABLE <táblanév>
  (<oszlopnév> <típus> [NOT NULL]
  [, <oszlopnév> <típus> [NOT NULL] , ...]
);
```

paranccsal lehet létrehozni. A lehetséges adattípusok implementációnként változhatnak, általában a következő adattípusok megtalálhatók:

`CHAR(n)`	max. *n* karakter hosszú szöveg (karaktersorozat);
`LONG`	mint a CHAR, de hosszára nincs felső korlát (nagyon nagy);
`NUMBER(n)`	az előjellel együtt, max n karakter széles egész szám;
`DATE`	dátum (és általában időpont)

Ha valamely oszlop definíciója tartalmazza a `NOT NULL` módosítót, a megfelelő mezőben mindig valamilyen legális értéknek kell szerepelnie.
A SCOTT demo adatbázis néhány tábláját (kisebb módosításokkal) a következőképpen lehetne létrehozni:

```
CREATE TABLE customer (
  custid      NUMBER (6) NOT NULL,
  name        CHAR (45),
  address     CHAR (40),
  city        CHAR (30),
  state       CHAR (2),
  zip         CHAR (9),
  area        NUMBER (3),
  phone       CHAR (9),
  repid       NUMBER (4),
  creditlimit NUMBER (9,2),
  comments    LONG);
CREATE TABLE ord  (
  ordid       NUMBER (4) NOT NULL,
  orderdate   DATE,
  commplan    CHAR (1),
  custid      NUMBER (6) NOT NULL,
```

```
  shipdate    DATE,
  total       NUMBER (8,2));
CREATE TABLE item  (
  ordid       NUMBER (4) NOT NULL,
  itemid      NUMBER (4) NOT NULL,
  prodid      NUMBER (6),
  actualprice NUMBER (8,2),
  qty         NUMBER (8),
  itemtot     NUMBER (8,2));
CREATE TABLE product (
  prodid      NUMBER (6),
  descrip     CHAR (30),
  partof      NUMBER (6),
  comments    LONG);
CREATE TABLE price (
  prodid      NUMBER (6)  NOT NULL,
  stdprice    NUMBER (8,2),
  minprice    NUMBER (8,2),
  startdate   DATE,
  enddate     DATE),
  currency_code CHAR (3));
CREATE TABLE currency (
  currency_code CHAR (3) NOT NULL,
  currency_name CHAR (45),
  is_european CHAR (1));
```

4.2. Táblák törlése

Táblát törölni a

```
DROP TABLE <táblanév>;
```

utasítással lehet.

5. Adatok bevitele, törlése, módosítása

5.1. Adatok bevitele

A `CREATE TABLE` utasítással létrehozott táblák kezdetben üresek. Új sort a következő utasítással lehet felvenni:

```
INSERT INTO <táblanév> [(<oszlopnév> [, <oszlopnév>, ...])]
      VALUES (<kif1> [, <kif2> , ...]);
```

Amennyiben nem adjuk meg az oszlopok nevét, akkor a – tábla deklarálásánál megadott sorrendben – minden mezőnek értéket kell adni, ha viszont megadtuk az egyes oszlopok neveit, akkor csak azoknak adunk értéket, mégpedig a felsorolásuk sorrendjében, a többi mező `NULL` értékű lesz.
Az egyes mezőknek `NULL` értéket is adhatunk, ha a deklaráció alapján az adott mezőnek lehet `NULL` értéke.
Figyelem: Amennyiben olyan oszlopnak akarunk `NULL` értéket adni, amelynek nem lehet `NULL` értéke, úgy a parancsvégrehajtás hibával leáll!
Egy ilyen paranccsal egyszerre csak egy sort tudunk felvenni a táblába.
Két új termék bevitele pl. az alábbi SQL utasítással lehetséges:

```
INSERT INTO product (prodid, descrip)
      VALUES (111111, 'Gozeke');
INSERT INTO product
      VALUES (111112, 'Oracle 6.0', NULL,
      'Relacios adatbazis-kezelo');
```

5.2. Adatok törlése

Sorokat kitörölni a

```
DELETE FROM <táblanév>
[WHERE <logikai kifejezés>];
```

paranccsal lehet.
Ha a WHERE hiányzik, akkor a tábla valamennyi sorát, egyébként csak a logikai kifejezés által kiválasztott sorokat törli.
Ha az előzőekben felvett adatok közül a 111112 azonosítójút (prodid) törölni akarjuk, akkor ezt a következő módon lehetséges:

```
DELETE FROM product
WHERE prodid = 111112;
```

5.3. Adatok módosítása

Sorokban az egyes mezők értékeit az

```
UPDATE <táblanév>
SET <oszlopnév> = <kifejezés> [,<oszlopnév> = <kifejezés> , ...]
[WHERE <logikai kifejezés>];
```

paranccsal lehet módosítani.
Ha a `WHERE` hiányzik, akkor a parancs a tábla valamennyi sorában módosít, egyébként csak a logikai kifejezés által kiválasztott sorokban.
Ha a PRODUCT táblában a "Gozeke"-hez megjegyzést akarunk fűzni, akkor ezt a következőképpen lehet megoldani:

```
UPDATE product
SET comments = 'mezogazdasagi gep'
WHERE descrip = 'Gozeke';
```

6. Lekérdezések

A lekérdezések általános szintaxisa a következő:

```
SELECT <jellemzők>
FROM <táblák>
[WHERE <logikai kifejezés>]
[<csoportosítás>]
[<rendezés>];
```

A lekérdezés művelete eredményül egy újabb táblát állít elő – persze lehet, hogy az eredménytáblának csak egy oszlopa és csak egy sora lesz. Az eredménytábla a lekérdezés után megjelenik vagy a tábla felhasználható egy másik parancsba beágyazva (pl. halmazműveletek).
A <*jellemzők*> definiálják az eredménytábla oszlopait,
a <*táblák*> adják meg a lekérdezésben résztvevő táblák nevét,
a <*logikai kifejezés*> segítségével "válogathatunk" az eredmény sorai között,
a <*csoportosítás*> az eredménytábla sorait rendezi egymás mellé,
a <*rendezés*> a megjelenő sorok sorrendjét határozza meg.
Nézzük meg, hogy a lekérdezés műveletével hogyan lehet megvalósítani a relációs algebra alapműveleteit.

6.1. Vetítés (projection)

```
SELECT <jellemzők> FROM <táblanév>;
```

A vetítés művelete egy táblából adott oszlopokat válogat ki. A <jellemzők> között kell felsorolni a kívánt oszlopokat.
Például ha a termékek kódjára és nevére vagyunk kíváncsiak:

```
SELECT prodid, descrip FROM product;
```

minden oszlop kiválasztása:

```
SELECT * FROM product;
```

Ha a *<jellemzők>*-ben csak a *-ot adjuk meg, akkor az adott tábla minden oszlopát kiválasztja. (Az egész táblát megjeleníti.)
A *<jellemzők>* közé nemcsak a *FROM* mögött megadott tábla oszlopainak nevét lehet megadni, hanem használhatjuk az SQL beépített műveleteit is: pl. egyszerű aritmetikai kifejezéseket új érték előállítására, oszlopfüggvényeket (lásd később!).
Az áfás ár számítása:

```
SELECT prodid, startdate, 1.25*stdprice FROM price;
```

A lekérdezésben az 1.25*stdprice-nak külön nevet is adhatunk az AS kulcsszó segítségével (oszlopszinonima), amelyre az ORDER BY (rendezés, ld. 6.7) klózban, vagy beágyazott lekérdezés esetén a beágyazó kontextusban hivatkozhatunk. Például:

```
SELECT prodid, startdate, 1.25*stdprice AS pricewithtax FROM price;
```

A kiválasztott oszlopokat tartalmazó táblákban lehetnek azonos sorok, ami ellentmond a relációs táblák egyik alapvető követelményének. Ennek ellenére a *SELECT* utasítás nem szűri ki automatikusan az azonos sorokat, mert ez túlságosan időigényes művelet. A programozónak kell tudnia, hogy az előállított táblákban lehetnek-e (zavarnak-e) ilyen sorok. Ha kell, a következőképpen szűrhetjük ki ezeket:

Az összes különböző terméknév:

```
SELECT DISTINCT descrip FROM product;
```

6.2. Szelekció (selection)

A szelekció műveleténél a tábla sorai közül válogatunk a `WHERE` segítségével. A *<logikai kifejezés>* igaz értékeinek megfelelő sorok kerülnek be az eredménytáblába.
A kifejezések elemi összetevői:
literálok különböző típusú értékekre: számok, szöveg, dátum;
oszlopok nevei;
a fenti elemekből elemi adatműveletekkel képzett kifejezések

számoknál	aritmetikai műveletek, aritmetikai függvények;
szövegeknél	`SUBSTR(), INSTR(), UPPER(), LOWER(), SOUNDEX(), ...;`
dátumoknál	+, -, konverziók;
halmazok	pl.: `(10,20,30)`;

zárójelek között egy teljes `SELECT` utasítás (egymásba ágyazott lekérdezések).
A fenti műveletekkel képzett adatokból logikai értéket a következő műveletekkel állíthatunk elő:

relációk	<, <=, =, !=, >=, >;
intervallumba tartozás	`BETWEEN` ... `AND` ...;
NULL érték vizsgálat	`IS NULL`, `IS NOT NULL`;

halmaz eleme	`IN <halmaz>`;
szövegvizsgálat mintával összevetés	`... LIKE <minta>`, ahol `%` a tetszőleges, akár nulla hosszúságú karaktersorozat, _ a tetszőleges, pontosan egy karakter jelzése.

Végül a logikai értékeket a zárójelezéssel, illetve az `AND`, `OR` és `NOT` műveletekkel lehet tovább kombinálni.
A 2000 dollárnál magasabb árak:

```
SELECT prodid, startdate, stdprice FROM price
WHERE stdprice > 2000;
```

A 2000 dollárnál magasabb áfás árak növekvő sorrendben:

```
SELECT prodid, startdate, 1.25*stdprice AS pricewithtax
FROM price
WHERE 1.25*stdprice > 2000
ORDER BY pricewithtax;
```

Az 1994. március 8.-án érvényes árak:

```
SELECT prodid, stdprice FROM price
WHERE '08-mar-94' BETWEEN startdate AND NVL(enddate,'31-dec-94');
```

Az `NVL(<oszlopnév>,<érték>)` azt biztosítja, hogy amennyiben `enddate` értéke `NULL` lenne, akkor azt `31-dec-94`-gyel helyettesíti.
A 2000 dollárnál alacsonyabb árak, ahol nincs minimális ár:

```
SELECT prodid, startdate, stdprice FROM price
WHERE stdprice < 2000 AND minprice IS NULL;
```

6.3. Illesztés (join)

A természetes illesztés műveleténél két vagy több tábla soraiból hozunk össze egy-egy új rekordot akkor, ha a két sor egy-egy mezőjének értéke megegyezik. A `SELECT` kifejezésben a <táblák>-ban kell megadni az érintett táblák neveit (vesszővel elválasztva), a `WHERE` mögötti logikai kifejezés definiálja azokat az oszlopokat, amely értékei szerint történik meg az illesztés.
Az egyes termékek neve, árai és az árak érvényességi ideje:

```
SELECT product.descrip, price.*
FROM product, price
WHERE product.prodid=price.prodid;
```

Látható, hogy mindkét felhasznált táblában azonos az illesztést megvalósító oszlop neve, a `WHERE`-t követő logikai kifejezésben az oszlop neve mellé meg kell adni a tábla nevét is. Hasonló helyzet előfordulhat a `SELECT`-et követő <*jellemzők*> között is.
Előfordulhat, hogy az ilyen lekérdezésben egyes sorok nem jelennek meg, mert az adott sorhoz a másik táblában nem található illeszthető sor. Ez lehet nem kívánatos eredmény, azonban az SQL-ben lehetőség van arra is, hogy ezeket a sorokat is illesszük, azaz az összekapcsolás során a másik táblához hozzárendelünk egy üres sort is. Ezt külső illesztés hívjuk.
A módosított példa a következőképpen néz ki:

```
SELECT product.descrip, price.stdprice, price.startdate
FROM product, price
WHERE product.prodid = price.prodid(+);
```

A (+) jelzi azt a táblát, amelyikben ha nincs a másik táblához illeszkedő sor, akkor is kell egy üres mezőket tartalmazó sort hozzávenni a másik táblához.
Külső illesztésnél a (+) jelet ki kell tenni a NULL értékekkel kiegészített tábla WHERE klózban szereplő valamennyi előfordulása után, kivéve, ha az adott feltétel egy újabb külső illesztést ír le. Ilyenkor a „kiegészítendő" táblát nem szabad (+) jellel jelölni. Az alábbi példa egy product -> price -> currency külső illesztés-láncot mutat. Az illesztési feltételekben csak a kiegészítő táblát jelöltük (+) jellel, míg a kiegészítendő tábla jelöletlen, l. az (1) kifejezés bal oldalán. A szűrési feltételekben a külső táblák összes előfordulása jelölendő, l. a (2) kifejezés bal oldalán.

```
SELECT product.descrip, price.stdprice, price.startdate
FROM product, price, currency
WHERE -- illesztési feltételek
      product.prodid = price.prodid(+)
   -- (1) a price táblánál nincs (+)
  AND price.currency_code = currency.currency_code(+)
   -- szűrések
  AND product.partof = 'fülke'
   -- (2) a price táblánál is van (+)
  AND price.minprice(+) > 300
  AND currency.is_european(+) = 'Y'
;
```

Az illesztésnél lehet ugyanarra a táblára többször is hivatkozni.
Az azonos nevű termékek (páronként):

```
SELECT a.descrip, a.prodid, b.prodid
FROM product a, product b
WHERE a.descrip = b.descrip AND a.prodid < b.prodid;
```

A többször használt tábla oszlopainak megkülönböztetésére a táblákat a FROM részben lokális névvel látjuk el. Lokális neveket természetesen különböző táblák esetén is használhatunk.
Az egyesítés mellett egyidejűleg más logikai kifejezéseket is használhatunk.

6.4. Oszlopfüggvények

A lekérdezés eredményeként előálló táblák egyes oszlopaiban lévő értékeken végrehajthatunk a szokásos nyelven ciklussal kifejezhető műveleteket, amelyek egyetlen értéket állítanak elő. Ilyenek:

`AVG()`	átlag,
`SUM()`	összeg,
`COUNT()`	darabszám,
`MAX()`	maximális érték,
`MIN()`	minimális érték

A 1994 január 1-től induló árak átlaga:

```
SELECT AVG(stdprice) FROM price WHERE startdate = '01-jan-1994';
```

Hány termék van:

```
SELECT COUNT(*) FROM product;
```

Hány különböző nevű termék van:

```
SELECT COUNT(DISTINCT descrip) FROM product;
```

Átlagos minimális ár:

```
SELECT AVG(NVL(minprice, stdprice)) FROM price;
```

Ha nem oszlopfüggvény eredményéből (és nem konstansból) származó érték is része az eredménytáblának, akkor csoportosítani kell ezen értéklista szerint. A csoportosítás szükséges akkor is, ha a programozó biztos benne: az összes kiválasztott sorban azonosak az értékek.
Például írhatnánk:

```
SELECT COUNT(*), AVG(stdprice) FROM price;
```

de hibás

```
SELECT COUNT(*), descrip FROM product;
```

Az alábbi lekérdezés annak ellenére hibás, hogy a WHERE feltétel garantálja: a startdate értéke minden sorban azonos.

```
SELECT startdate, AVG(stdprice) FROM price
WHERE startdate = '01-jan-94';
```

Ehelyett írhatnánk például:

```
SELECT startdate, AVG(stdprice) FROM price
WHERE startdate = '01-jan-94'
GROUP BY startdate;
```

(a GROUP BY jelentését ld. az 6.6 szakaszban)

6.5. Egymásba ágyazott lekérdezések

A `WHERE` utasítás mögött állhat egy teljes `SELECT` utasítás is.
Az 1994. utáni árral rendelkező termékek listája:

```
SELECT prodid, descrip FROM product;
WHERE prodid IN
  (SELECT prodid FROM price
   WHERE startdate >= '01-jan-94');
```

Azaz először kiválasztjuk az árak táblából azon termékazonosítókat, amelyekhez 1994-es dátum tartozik, majd azt vizsgáljuk, hogy az ezekből képzett halmazban található-e az adott termék azonosítója. Mellesleg ugyanezt a listát megkaphatnánk az egyesítés műveletével is:

```
SELECT prodid, product.descrip
FROM product, price
WHERE product.prodid = price.prodid
AND price.startdate >= '01-jan-94';
```

Vegyük észre, hogy a kettő mégsem teljesen ekvivalens: ha egy terméknek az ára 1994-ben megváltozott, akkor az egyesítéssel való lekérdezés minden ár-bejegyzéshez generál egy sort, ellentétben beágyazott lekérdezés fenti formájával.
A beágyazott lekérdezés vagy egyetlen értéket – azért mert egyetlen megfelelő sor egyetlen oszlopát választottuk ki, illetve oszlopfüggvényt használtunk –, vagy több értéket – több sort – állít elő. Az előbbi esetben a `SELECT` értékét az elemi értékekkel azonos módon használhatjuk. Több érték egy halmazt jelent, tehát a halmazműveleteket használhatjuk. A korábban megismert `IN()` – eleme – művelet mellett használható az `ANY()` illetve az `ALL()` művelet, ahol a kívánt reláció a halmaz legalább egy, illetve valamennyi értékére igaz.
Legmagasabb árú termékek (lehet, hogy több van!):

```
SELECT prodid, stdprice FROM price
WHERE stdprice >= ALL (SELECT stdprice FROM price);
```

illetve ugyanez a példa oszlopfüggvény felhasználásával:

```
SELECT prodid, stdprice FROM price
WHERE stdprice = (SELECT MAX(stdprice) FROM price);
```

6.6. Csoportosítás (grouping)

Az oszlopfüggvények a teljes kiválasztott táblára – minden sorra – lefutnak. Gyakran célszerű lenne a kiválasztott sorokat valamilyen szempont szerint csoportosítani és az oszlopfüggvényeket az egész tábla helyett ezekre a csoportokra alkalmazni.
Legmagasabb ár ugyanazon naptól:

```
SELECT startdate, MAX(stdprice) FROM price
GROUP BY startdate;
```

Természetesen az oszlopfüggvények használatához hasonlóan a `SELECT` <*jellemzők*> között csak a csoportosítás alapját képező oszlop neve, illetve a csoportokra alkalmazott oszlopfüggvények szerepelhetnek.
A csoportosítás után az eredményből bizonyos csoportok kihagyhatók.
Maximális árak azonos napon az 1000 és 3000 dollár közötti tartományban:

```
SELECT startdate, MAX(stdprice) AS maxprice FROM price
GROUP BY startdate
HAVING MAX(stdprice) BETWEEN 1000 AND 3000;
```

A `HAVING` mögött természetesen csak egy-egy közös jellemzőire vonatkozó értékek – a csoportosítás alapját képező oszlop értéke, vagy oszlopfüggvények eredménye – szerepelhet. Természetesen a csoportosítás előtt azért a `WHERE` feltételek használhatók. Célszerű – gyorsabb – `WHERE` feltételeket alkalmazni mindenhol, ahol csak lehet, és a `HAVING` szerkezetet csak akkor alkalmazni, amikor a teljes csoporttól függő értékeket akarjuk vizsgálni.

6.7. Rendezés (sorting)

Az eddig tárgyalt lekérdezések eredményében a sorok „véletlenszerű” – a programozó által nem megadható – sorrendben szerepeltek. A sorrendet az `ORDER BY` által megadott rendezéssel lehet szabályozni. A rendezés több oszlop értékei szerint is történhet, ilyenkor az először megadott oszlop szerint rendezünk, majd az itt álló azonos értékek esetében használjuk a következőnek megadott oszlop(ok) értékét. Minden egyes oszlop esetében külön meg lehet adni a rendezés „irányát”, amely alapesetben emelkedő, de a `DESC` módosítóval csökkenő rendezés írható elő.
Az 111111-es termék ára (időrendben):

```
SELECT stdprice, startdate FROM price WHERE prodid = 111111
ORDER BY startdate;
```

Minden termék valaha elért legmagasabb ára csökkenő sorrendben:

```
SELECT prodid, MAX(stdprice) FROM price
GROUP BY prodid
ORDER BY MAX(stdprice) DESC;
```

6.8. Halmazműveletek

Az egyes lekérdezések által előállított táblák halmazként is felfoghatók, az SQL nyelv ezen táblák kombinálására tartalmaz halmazműveleteket is. Ilyenek:

`UNION`	unió,
`INTERSECT`	metszet,
`MINUS`	különbség,
`IN`	tartalmazás

A műveleteket két `SELECT` utasítás közé kell írni. (Lásd korábban az egymásba ágyazott lekérdezéseknél!)

6.9. Hierarchikus kapcsolat lekérdezés

A relációs táblák segítségével le tudunk írni hierarchikus kapcsolatokat a különböző sorok között.
Például a

```
SELECT descrip, prodid, partof
FROM product
CONNECT BY PRIOR prodid = partof
START WITH descrip = 'gozeke';
```

utasítás kiírja a gőzeke alkatrészeit az utolsó kerékig lebontva.
A CONNECT BY klózban megadott feltételt kielégítő rekordpárok a hierarchiában szülő-gyerek viszonyban vannak. A PRIOR kulcsszóval jelölt kifejezés a szülő rekordon értelmezett. Így a példában a PRIOR `prodid` pl. a fülke azonosítója, míg a fülkeajtó rendszerbe illeszkedését írja le a `partof` attribútum.

6.10. Egyéb nem szorosan az SQL nyelvhez tartozó utasítások

Nem tartoznak szorosan az SQL nyelvhez, de a legtöbb rendszer tartalmaz olyan utasításokat, amelyekkel a lekérdezések által előállított táblázatok megjelenését – pl.: az oszlopok neveit, szélességét, adatformátumát, illesztését, tördelését stb. – definiálhatjuk. Lásd Server SQL Language Reference Manual helpet!

7. Nézetek

A nézetek olyan virtuális táblák, amelyek a fizikai táblákat felhasználva a tárolt adatok más és más logikai modelljét, csoportosítását tükrözik. Nézetek a

```
CREATE VIEW <nézetnév> [(<oszlopnév> [, <oszlopnev>, ...])]
AS <lekérdezés>;
```

paranccsal készíthetők.
A lekérdezésre az egyedüli megkötés, hogy rendezést nem tartalmazhat. Amennyiben nem adunk meg oszlopneveket, a nézetek oszlopai a `SELECT` után felsorolt oszlopok neveivel azonosak. Meg kell viszont adni az oszlopneveket, ha a `SELECT` számított értékeket is előállít.
Például az alábbi nézet megmutatja minden termék aktuális árát:

```
CREATE VIEW prods AS
  SELECT product.prodid, product.descrip pdescrip,
    x.stdprice sprice, x.minprice mprice
  FROM product, price x
  WHERE x.prodid = product.prodid
  AND x.startdate >= all (
    SELECT startdate
    FROM price i
    WHERE i.prodid = x.prodid);
```

A nézetek a lekérdezésekben a táblákkal megegyező módon használhatók. Jelentőségük, hogy az adatok más modelljét fejezik ki, felhasználhatók a tárolt információ részeinek elrejtésére, pl. különböző felhasználók más-más nézeteken keresztül szemlélhetik az adatokat. A nézet általában csak olvasható, az adatmódosító műveletekben csak akkor szerepelhet, ha egyetlen táblából keletkezett és nem tartalmaz számított értékeket.
Nézetet törölni a

```
DROP VIEW <nézetnév>;
```

utasítással lehet.
A fenti nézetet az alábbi utasítással törölhetjük:

```
DROP VIEW prods;
```

8. Indexek

Az indexek a táblákban való keresést gyorsítják meg.

8.1. Indexek létrehozása

Egy indexet a

```
CREATE [UNIQUE] INDEX <indexnév>
ON <táblanév> (<oszlopnév> [, <oszlopnév> , ...]);
```

utasítással lehet létrehozni.
Az indexeket az adatbázis-kezelő a táblák minden módosításnál felfrissíti. Amennyiben valamelyik indexet az `UNIQUE` kulcsszóval definiáltuk, a rendszer biztosítja, hogy az adott oszlopban minden mező egyedi értéket tartalmaz. Lehetséges több oszlopot egybefogó, kombinált indexek létrehozása is. Az indexek a létrehozásuk után a felhasználó számára láthatatlanok, csak éppen bizonyos lekérdezéseket gyorsítanak. Indexeket azokra az oszlopokra érdemes definiálni, amelyek gyakran szerepelnek keresésekben. Index nélkül minden kéréshez be kell olvasni az egész táblát.
Ha a keresési feltételhez van megfelelő index, akkor az adatbázis-kezelő a diszkről csak a valóban szükséges sorokat olvassa be.
Például a

```
SELECT * FROM emp WHERE ename = 'JONES';
```

az `emp` táblában keresés nélkül kiválaszthatja `JONES` rekordját, ha az `ename` oszlopra definiáltunk indexet. Az indexek akkor is gyorsítják a keresést, ha csak a keresési feltétel egy részére vonatkoznak.

8.2. Indexek törlése

Az indexeket a

```
DROP INDEX <indexnév>;
```

paranccsal törölhetjük.

9. Jogosultságok definiálása

Az egyes felhasználók részint az adatbázis-kezelő rendszerrel, részint az egyes objektumaival különböző műveleteket végezhetnek. Ezeknek a megadására szolgálnak a GRANT utasítások.
A

```
GRANT [DBA | CONNECT | RESOURCES]
TO <felhasználó> [, <felhasználó> , ...]
IDENTIFIED BY <jelszó> [, <jelszó>, ...];
```

paranccsal az egyes felhasználóknak az adatbázishoz való hozzáférési jogát szabályozzák. A `DBA` jogosultság az adatbázis adminisztrátorokat (DataBase Administrator) definiálja, akiknek korlátlan jogai vannak az összes adatbázis objektum felett, nemcsak létrehozhatja, módosíthatja, illetve törölheti, de befolyásolhatja az objektumok tárolásával, hozzáférésével kapcsolatos paramétereket is. A `RESOURCES` jogosultsággal rendelkező felhasználók létrehozhatnak, módosíthatnak, illetve törölhetnek új objektumokat, míg a `CONNECT` jogosultság csak az adatbázis-kezelőbe belépésre jogosít.
Az egyes objektumokhoz – táblák, illetve nézetek – a hozzáférést a

```
GRANT <jogosultság> [, <jogosultság> , ...]
ON <tábla vagy nézetnév>
TO <felhasználó>
[WITH GRANT OPTION];
```

parancs határozza meg. A *<jogosultság>* az objektumon végezhető műveleteket adja meg, lehetséges értékei:

```
ALL,
SELECT,
INSERT,
UPDATE,
DELETE,
ALTER,
INDEX
```

Az utolsó két művelet nézetekre nem alkalmazható. A felhasználó neve helyett PUBLIC is megadható, amely bármelyik felhasználóra vonatkozik. A WITH GRANT OPTION-nel megkapott jogosultságokat a felhasználók tovább is adhatják.

10. Tábladefiníciók módosítása

Már létező táblákat módosítani az

```
ALTER TABLE <táblanév>
[ADD | MODIFY] <oszlopnév> <típus>;
```

paranccsal lehet, ahol `ADD` egy új, `NULL` értékű oszlopot illeszt a táblához, míg `MODIFY` paranccsal egy létező oszlop típusának tulajdonságait módosítjuk.
Például egy új oszlop felvétele a mytable táblába:

```
ALTER TABLE mytable
ADD id NUMBER(6);
```

11. Tranzakciók[15]

Az adatbázisok módosítása általában nem történhet meg egyetlen lépésben, hiszen legtöbbször egy módosítás során több táblában tárolt információn is változtatni akarunk, illetve egyszerre több rekordban akarunk módosítani, több rekordot akarunk beilleszteni. Előfordulhat, hogy módosítás közben meggondoljuk magunkat, vagy ami még súlyosabb következményekkel jár, hogy az adatbázis-kezelő leáll. Ilyenkor a tárolt adatok inkonzisztens állapotba kerülhetnének, hiszen egyes módosításokat már elvégeztünk, ehhez szorosan hozzátartozó másokat viszont még nem.
A tranzakció az adatbázis módosításának olyan sorozata, amelyet vagy teljes egészében kell végrehajtani, vagy egyetlen lépését sem. Az adatbázis-kezelők biztosítják, hogy mindig vissza lehessen térni az utolsó teljes egészében végrehajtott tranzakció utáni állapothoz.
Egy folyamatban lévő tranzakciót vagy a `COMMIT` utasítással zárhatjuk le, amely a korábbi `COMMIT` óta végrehajtott összes módosítást véglegesíti, vagy a `ROLLBACK` utasítással törölhetjük a hatásukat, visszatérve a megelőző `COMMIT` kiadásakor érvényes állapotba.
Beállítható, hogy az SQL műveletek automatikusan `COMMIT` műveletet hajtsanak végre:

```
SET AUTOCOMMIT [ON | OFF];
```

Az `ON` állapotban minden SQL utasítás, `OFF` állapotban az `ALTER, CREATE, DROP, GRANT` és `EXIT` utasítások sikeres végrehajtása `COMMIT`-ot is jelent. (Azaz ezeket nem lehet visszagörgetni, tehát pl. törölt tábla nem állítható vissza!)
A rendszer hardverhiba utáni újraindításkor, illetve hibás `INSERT, UPDATE` vagy `DELETE` parancs hatására automatikusan `ROLLBACK`-et hajt végre. Érdemes hát biztonságos helyeken `COMMIT` parancsokat kiadni, nehogy egy hibásan kiadott parancs automatikusan visszavonjon korábbi módosításokat.

[15] Ld. az Adatbázisok tárgy kapcsolódó előadásait is (kb. november elejétől)

12. Párhuzamos hozzáférés szabályozása

Az adatbázis-kezelő rendszereket tipikusan több felhasználó használja, ezzel kapcsolatban újabb problémák merülnek fel, ezért az egyes táblákhoz a párhuzamos hozzáférést külön-külön lehet szabályozni:

```
LOCK TABLE <táblanév> [, <táblanév> , ...]
IN [SHARE | SHARED UPDATE | EXCLUSIVE] MODE [NOWAIT];
```

A `LOCK` paranccsal egy felhasználó megadhatja, hogy az egyes táblákhoz más felhasználónak milyen egyidejű hozzáférést engedélyez. Az utasítás végrehajtásánál a rendszer ellenőrzi, hogy a `LOCK` utasításban igényelt felhasználási mód kompatibilis-e a táblára érvényben lévő kizárással. Amennyiben megfelelő, az utasítás visszatér és egyéb utasításokat lehet kiadni. Ha az igényelt kizárás nem engedélyezett, az utasítás várakozik, amíg az érvényes kizárást megszüntetik, ha a parancs nem tartalmazza a `NOWAIT` módosítót. Ebben az esetben a `LOCK` utasítás mindig azonnal visszatér, de visszaadhat hibajelzést is.
A táblához a hozzáférést az első sikeres `LOCK` utasítás definiálja.
Az `EXCLUSIVE` mód kizárólagos hozzáférést biztosít a táblához. A `SHARE` módban megnyitott táblákat mások olvashatják, a `SHARE UPDATE` módban mások módosíthatják is, ilyenkor a kölcsönös kizárást az ORACLE soronként biztosítja (automatikusan), tehát nem írhatják ketten egyidejűleg ugyanazt a sort.

13. Konzisztenciafeltételek

A táblák definíciójánál eddig csak azt adhattuk meg, hogy milyen adattípusba tartozó értékeket lehet az egyes oszlopokban használni, illetve mely oszlopokban kell feltétlenül értéknek szerepelnie. Célszerű lenne a táblákhoz olyan feltételeket rendelni, amelyek szigorúbb feltételeket definiálnak az egyes adatokra, amelyeket aztán a rendszer a tábla minden módosításánál ellenőriz (ld. Függelék: Adatbázis kényszerek az Oracle-ben). Ilyenek például:

- az egyes adatok értékkészletének az általános adattípusnál pontosabb definíciója (pl. adott intervallumba tartozás, adott halmazba tartozás, ahol a halmaz lehet egy másik tábla egyik oszlopának értékei);
- az oszlop elsődleges kulcs, azaz a tábla soraiban minden értéke különböző (hasonló hatás elérhető a `UNIQUE` indexszel is);
- az oszlop idegen kulcs, azaz értéke meg kell, hogy egyezzen egy másik tábla elsődleges kulcs oszlopának valamelyik létező elemével.

Amennyiben a táblák módosításánál valamelyik feltételt megsértenénk, a rendszer egy kivételt (exception) generál és lefuttatja a hibához tartozó kiszolgáló utasítást, ha van ilyen.
Az SQL nyelv további elemeiről az online helpben található leírás.

III. labor: Java Database Connectivity (JDBC)[16]

Szerzők: Mátéfi Gergely, Kollár Ádám, Remeli Viktor, Kamarás Roland, Varsányi Márton

1. Bevezetés

A Java Database Connectivity (JDBC) a Javából történő adatbázis elérés gyártófüggetlen *de facto* szabványa. A JDBC programozói felület (Application Programming Interface, API) Java osztályok és interfészek halmaza, amelyek relációs adatbázisokhoz biztosítanak alacsonyszintű hozzáférést: kapcsolódást, SQL utasítások végrehajtását, a kapott eredmények feldolgozását. Az interfészeket a szállítók saját *meghajtói (driverei)* implementálják. A meghajtóknak – a Java működésének megfelelően – elegendő futási időben rendelkezésre állniuk, így a programfejlesztőnek lehetősége van a Java alkalmazást az adatbázis-kezelő rendszertől (DBMS-től) függetlenül elkészítenie.

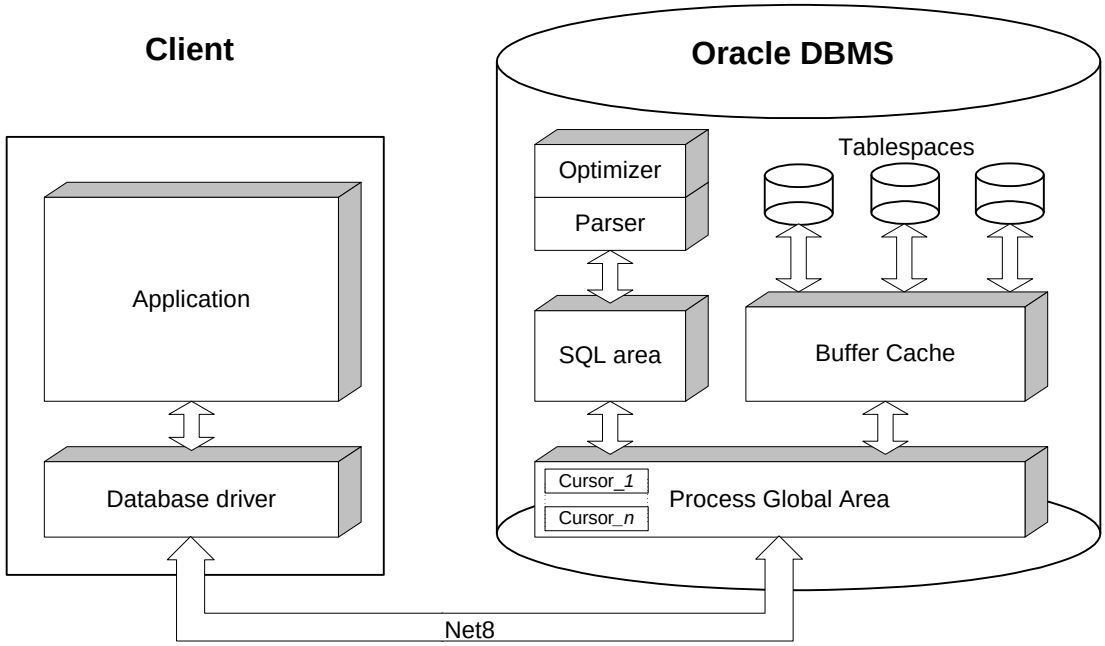
Jelen labor célja Java és JDBC környezeten keresztül az adatbázisalapú, kliens-szerver architektúrájú alkalmazások fejlesztésének bemutatása. Az első alfejezetben a kliens-szerver architektúrát mutatjuk be röviden, ezt követően a JDBC legfontosabb nyelvi elemeit foglaljuk össze, végül konkrét példán demonstráljuk a JDBC használatát. A segédlet a laborkörnyezet adottságaihoz igazodva a JDBC 1.2 változatú API bemutatására korlátozódik.

2. Adatbázis-kezelés kliens-szerver architektúrában[17]

Kliens-szerver architektúra mellett a kliensen futó alkalmazás hálózaton keresztül, a gyártó által szállított meghajtó segítségével éri el a DBMS-t. A meghajtó az alkalmazástól függően lehet például C nyelvű könyvtár, ODBC- vagy JDBC- meghajtó.

[16] Ld. még a segédlet végén a reguláris kifejezésekről szóló függeléket is.

[17] Az alfejezet az alapfogalmakat az Oracle RDBMS (jelentősen leegyszerűsített) működésén keresztül mutatja be, maguk a fogalmak azonban nem Oracle-specifikusak



Az adatbázis műveleteket megelőzően a felhasználónak egy ún. *adatbázis-kapcsolatot* (*session*) kell felépítenie, melynek során autentikálja magát a DBMS felé. Oracle rendszerben a felépítés során a DBMS a session számára erőforrásokat allokál: memóriaterületet (Process Global Area, PGA) foglal és kiszolgálófolyamatot (server process) indít.[18]

A session élete során a kliens adatbázis műveleteket kezdeményezhet, melyeket a meghajtó SQL utasításként továbbít a DBMS felé. Az utasítást a DBMS több lépésben dolgozza fel. A feldolgozás kezdetén a kiszolgálófolyamat *memóriaterületet különít el* a PGA-ban: itt tárolódnak a feldolgozással kapcsolatos információk, többek között a lefordított SQL utasítás és az eredményhalmazbeli pillanatnyi pozíció is (ld. lejjebb). Az elkülönített memóriaterület leíróját *kurzornak* (*cursor*), a feldolgozás megkezdését a *kurzor megnyitásának* is nevezik. Egy session egy időben több megnyitott kurzorral is rendelkezhet.

A feldolgozás első lépése az SQL *utasítás elemezése* (*parsing*), melynek során a DBMS lefordítja az utasítást tartalmazó stringet, ezt követi az érintett adatbázis objektumokhoz tartozó *hozzáférési jogosultságok ellenőrzése*. A sikeresen lefordított utasításhoz az Optimizer készíti el az ún. *végrehajtási tervet* (*execution plan*). A végrehajtási terv tartalmazza az utasítás által érintett sorok fizikai leválogatásának lépéseit: mely táblából kiindulva, mely indexek felhasználásával, hogyan történjék a leválogatás. Mivel az elemzés és a végrehajtási terv meghatározás számításigényes művelet, a DBMS gyorsítótárban (*SQL area*) tárolja legutóbbi SQL utasítások végrehajtási tervét.

Egy adatbázisalapú alkalmazás futása során tipikusan néhány, *adott szerkezetű* SQL utasítást használ, de *eltérő paraméterezéssel*. Egy számlázószoftver például rendre ugyanazon adatokat hívja le az ügyfelekről, a lekérdezésekben mindössze az ügyfélazonosító változik. Az SQL nyelv lehetőséget teremt ezen utasítások paraméteres megírására:

```
SELECT NEV, CIM, ADOSZAM FROM UGYFEL WHERE UGYFEL_ID = ?
```

A paraméteres SQL utasítást az adatbázis-kezelő az első feldolgozáskor fordítja le, a későbbi meghívások során már nincs szükség újrafordításra. A gyorsítótár használatát az teszi lehetővé, hogy a feldolgozás során a *paraméterek behelyettesítése csak az elemzést és végrehajtási terv meghatározást követően történik.*

A végrehajtási terv meghatározása és az esetleges behelyettesítések után az SQL utasítás *végrehajtódik*. SELECT típusú lekérdezések esetén a kiválasztott sorok logikailag egy *ered-*

[18] Van lehetőség osztott szerverfolyamatok használatára is.

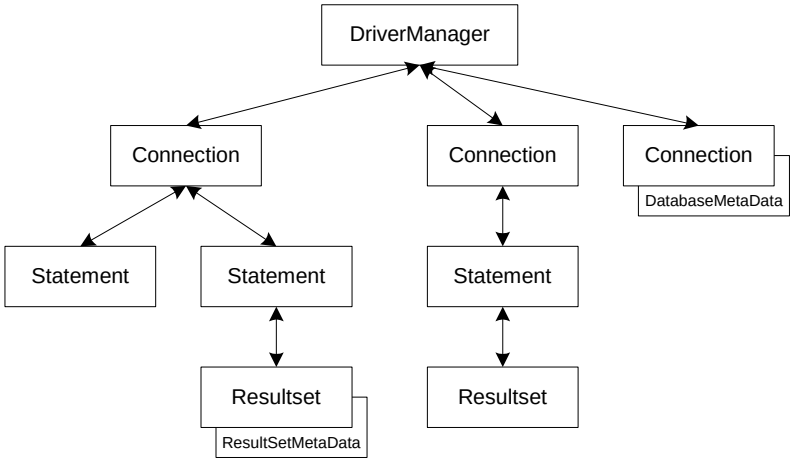
ménytáblát képeznek, melynek sorait a kliens egyenként[19] kérdezheti le az ún. *fetch* művelettel.
Az eredménytábla kiolvasása, illetve a tranzakció befejezése (commit/rollback) után a feldolgozásra elkülönített memóriaterület felszabadul, a *kurzor bezáródik*.

3. A JDBC 1.2 API

3.1. A programozói felület felépítése

A JDBC API Java osztályok és interfészek halmazából áll. A leglényegesebb osztályok és interfészek:

- java.sql.DriverManager osztály az adatbázis URL feloldásáért és új adatbázis kapcsolatok létrehozásáért felelős
- java.sql.Connection interfész egy adott adatbázis kapcsolatot reprezentál
- java.sql.DatabaseMetaData interfészen keresztül az adatbázissal kapcsolatos (meta)információkat lehet lekérdezni
- java.sql.Statement interfész SQL utasítások végrehajtását vezérli
- java.sql.ResultSet interfész egy adott lekérdezés eredményeihez való hozzáférést teszi lehetővé
- java.sql.ResultSetMetaData interfészen keresztül az eredménytábla metainformációi kérdezhetők le



3.2. Adatbázis kapcsolatok menedzsmentje

A JDBC meghajtók menedzsmentjét, új kapcsolatok létrehozását-lebontását a `java.sql.DriverManager` osztály végzi. A DriverManager tagváltozói és metódusai statikusak, így példányosítására nincs szükség az alkalmazásban. Egy új kapcsolat létrehozása a DriverManager-en keresztül egyetlen parancssorral elvégezhető:

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

A `getConnection` függvény első paramétere az adatbázist azonosító URL string, a második és harmadik paramétere a adatbázis felhasználót azonosító név és jelszó. Az URL tartalma adatbázisfüggő, struktúrája konvenció szerint a következő:

```
jdbc:<subprotocol>:<subname>
```

[19] A hatékony működés érdekében az adatbázis meghajtó egy fetch során kötegelten több sort is lehozhat.

ahol a <subprotocol> az adatbázis kapcsolódási mechanizmust azonosítja és a <subname> tag tartalmazza az adott mechanizmussal kapcsolatos paramétereket. Példaképpen a “Fred” által azonosított ODBC adatforráshoz a következő utasítással kapcsolódhatunk:

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

Ha a meghajtó által előírt URL már tartalmazza a felhasználói nevet és jelszót, akkor a függvény második és harmadik paramétere elmaradhat. A `getConnection` függvény meghívásakor a DriverManager egyenként lekérdezi a *regisztrált* JDBC meghajtókat, és az *első* olyan meghajtóval, amely képes a megadott URL feloldására, felépíti az adatbázis kapcsolatot. A kívánt műveletek elvégzése után a kapcsolatot a `Connection.close` metódusával kell lezárni. A `close` metódus a `Connection` objektum megsemmisítésekor (garbage collection) automatikusan is meghívódik.
A meghajtókat használatba vételük előtt *be kell tölteni* és *regisztrálni* kell a DriverManager számára. A programozónak általában csak a meghajtóprogram betöltéséről gondoskodnia, a meghajtók a statikus inicializátorukban rendszerint automatikusan regisztráltatják magukat. A betöltés legegyszerűbb módja a `Class.forName` metódus használata, például:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Biztonsági megfontolások miatt *applet* csak olyan meghajtókat használhat, amelyek vagy a lokális gépen helyezkednek el, vagy ugyanarról a címről lettek letöltve, mint az applet kódja. A „megbízhatatlan forrásból” származó appletekkel szemben a „megbízható” applikációkat teljes értékű programként futtatja a JVM, azokra a megkötés nem vonatkozik.

3.3. SQL utasítások végrehajtása

Egyszerű SQL utasítások végrehajtására a `Statement` interfész szolgál. Egy utasítás végrehajtásához az interfészt megvalósító meghajtó osztályt *példányosítani* kell, majd a példány – SQL utasítástól függő - *végrehajtó metódusát* kell meghívni. Egy `Statement` példány az aktív adatbázis-kapcsolatot reprezentáló `Connection` példány `createStatement` metódusával hozható létre:

```
Statement stmt = con.createStatement();
```

A `Statement` osztály háromféle végrehajtó metódussal rendelkezik:
`executeQuery`: a paraméterében megadott SQL lekérdezést végrehajtja, majd az eredménytáblával (`ResultSet`) tér vissza. A metódus lekérdező (SELECT) utasítások végrehajtására használandó.
`executeUpdate`: a paraméterében megadott SQL utasítást végrehajtja, majd a módosított sorok számával tér vissza. Használható mind adatmanipulációs (DML), mind adatdefiníciós (DDL) utasítások végrehajtására. DDL utasítások esetén a visszatérési érték 0.
`execute`: a paraméterében megadott SQL utasítást hajtja végre. Az előző két metódus általánosításának tekinthető. Visszatérési értéke true, ha az utasítás által visszaadott eredmény `ResultSet` típusú, ekkor az a `Statement.getResultSet` metódussal kérdezhető le. Az utasítással módosított sorok számát a `Statement.getUpdateCount` metódus adja vissza.
A következő példában a klasszikus Kávészünet Kft. számlázószoftveréhez hozzuk létre a számlák adatait tartalmazó táblát:

```
int n = stmt.executeUpdate("CREATE TABLE COFFEES ( " +
          "COF_NAME VARCHAR(32)," +
          "SUP_ID   NUMBER(8),  " +
          "PRICE    NUMBER(6,2)," +
          "SALES    NUMBER(4),  " +
          "TOTAL    NUMBER(6,2) )");
```

A vállalkozás beindulása után az alábbi utasítással tudjuk lekérdezni az eddigi vásárlásokat:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM COFFEES");
```

Egy utasítás végrehajtása akkor zárul le, ha az összes visszaadott eredménytábla fel lett dolgozva (minden sora kiolvasásra került). Az utasítás végrehajtása manuálisan is lezárható a `Statement.close` metódussal. Egy `Statement` objektum végrehajtó függvényének újbóli meghívása lezárja ugyanazon objektum korábbi lezáratlan végrehajtását.

A *paraméteres SQL utasítások* kezelése némiképp eltér az egyszerű SQL utasításokétól. A JDBC-ben a `PreparedStatement` interfész reprezentálja a paraméteres SQL utasításokat. Létrehozása az egyszerű `Statement`-hez hasonlóan, a kapcsolatot reprezentáló Connection példány `prepareStatement` metódusával történik, *a paraméteres SQL utasítás megadásával*:

```
PreparedStatement updateSales = con.prepareStatement(
  "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

A `PreparedStatement`-ben tárolt utasítás – kérdőjelekkel jelzett – paraméterei a `setXXX` metóduscsalád segítségével állíthatók be. A végrehajtásra a `Statement` osztálynál már megismert `executeQuery`, `executeUpdate` és `execute` metódusok használhatók, amelyeket itt *argumentum nélkül* kell megadni. A következő kódrészlet a COFFEES táblába történő adatfelvitelt szemlélteti, paraméteres SQL utasítások segítségével:

```
PreparedStatement updateSales;
String updateString =
  "update COFFEES set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
  updateSales.setInt(1, salesForWeek[i]);
  updateSales.setString(2, coffees[i]);
  updateSales.executeUpdate();
}
```

A `setXXX` metódusok két argumentumot várnak. Az első argumentum a beállítandó SQL paraméter indexe; a paraméterek az SQL utasításban balról jobbra, 1-gyel kezdődően indexelődnek. A második argumentum a beállítandó érték. A bemeneti paraméterek megadásánál a JDBC *nem végez implicit típuskonverziót*, a programozó felelőssége, hogy az adatbázis-kezelő által várt típust adja meg. Null érték a `PreparedStatement.setNull` metódussal állítható be. Egy beállított paraméter az SQL utasítás többszöri lefuttatásánál is felhasználható.

Tárolt eljárások és függvények meghívására a `CallableStatement` interfész használható. A `CallableStatement` a kapcsolatot reprezentáló `Connection` példány `prepareCall` metódusával példányosítható a korábbiakhoz hasonló módon. A `CallableStatement` interfész a `PreparedStatement` interfész leszármazottja, így a bemeneti (IN) paraméterek a `setXXX` metódusokkal állíthatók. A kimeneti (OUT) paramétereket a végrehajtás előtt a `CallableStatement.registerOutParameter` metódussal, a típus megadásával regisztrálni kell. A végrehajtást követően a `getXXX` metóduscsalád használható a kimeneti paraméterek lekérdezésére, mint a következő példa szemlélteti:

```
CallableStatement stmt = conn.prepareCall("call getTestData(?,?)");
stmt.registerOutParameter(1,java.sql.Types.TINYINT);
stmt.registerOutParameter(2,java.sql.Types.DECIMAL);
stmt.executeUpdate();
```

```
byte x = stmt.getByte(1);
BigDecimal n = stmt.getBigDecimal(2);
```

A `setXXX` metódusokhoz hasonlóan, a `getXXX` metódusok *sem végeznek típuskonverziót*: a programozó felelőssége, hogy az adatbázis által kiadott típusok, a `registerOutParameter` és a `getXXX` metódusok összhangban legyenek.

3.4. Eredménytáblák kezelése

A lekérdezések eredményeihez a `java.sql.ResultSet` osztályon keresztül lehet hozzáférni, melyet a `Statement` interfészek `executeQuery`, illetve `getResultSet` metódusai példányosítanak. A `ResultSet` által reprezentált eredménytáblának mindig az *aktuális* (kurzor által kijelölt) sora érhető el. A kurzor kezdetben mindig az első sor *elé* mutat, a `ResultSet.next` metódussal léptethető a következő sorra.[20] A `next` metódus visszatérési értéke false, ha a kurzor *túlment* az utolsó soron, true egyébként.
Az aktuális sor mezőinek értéke a `getXXX` metóduscsaláddal kérdezhető le.[21] A mezőkre a `getXXX` függvények kétféle módon hivatkozhatnak: oszlopindexszel, illetve az oszlopknevekkel. Az oszlopok az SQL lekérdezésben balról jobbra, 1-gyel kezdődően indexelődnek. Az oszlopnevekkel történő hivatkozás a futásidőben történő leképezés miatt kevésbé hatékony, ellenben kényelmesebb megoldást kínál. A `ResultSet.getXXX` metódusok, szemben a `PreparedStatement` és a `CallableStatement` `getXXX` függvényeivel, *automatikus típuskonverziót végeznek*. Amennyiben a típuskonverzió nem lehetséges (például a `getInt` függvény meghívása a VARCHAR típusú, "foo" stringet tartalmazó mezőre), `SQLException` kivétel lép fel. Ha a mező *SQL NULL értéket tartalmaz*, akkor a `getXXX` metódus zérus, illetve Java null értéket ad vissza, a `getXXX` függvénytől függően. A mező értékének kiolvasása *után* a `ResultSet.wasNull` metódussal ellenőrizhető, hogy a kapott érték SQL NULL értékből származott-e.[22]
Az eredménytábla lezárásával a programozónak általában nem kell foglalkoznia, mivel ez a `Statement` lezáródásával automatikusan megtörténik. A lezárás ugyanakkor manuálisan is elvégezhető a `ResultSet.close` metódussal.

Az eredménytáblákkal kapcsolatos metainformációkat a `ResultSetMetaData` interfészen keresztül lehet elérni. Az interfészt megvalósító objektumot a `ResultSet.getMetaData` metódus adja vissza. A `ResultSetMetaData getColumnCount` metódusa az eredménytábla oszlopainak számát, a `getColumnName(int column)` az indexszel megadott oszlop elnevezését adja meg.
A következő példa a `ResultSet` és a `ResultSetMetaData` használatát szemlélteti. A lekérdezés első oszlopa integer, a második String, a harmadik bájtokból alkotott tömb típusú:

```
Statement stmt = conn.CreateStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM table1”);
ResultSetMetaData rsmd = r.getMetaData();
for (int j = 1; j <= rsmd.getColumnCount(); j++) {
  System.out.print(rsmd.getColumnName(j));
}
System.out.println;
while (r.next()) {
  // Aktuális sor mezőinek kiíratása
  int i = r.getInt("a");
```

[20] Csak a JDBC 2.0 változatban van lehetőség van a kurzor visszafelé léptetésre és adott sorra történő mozgatására (ha ezt a meghajtó is támogatja)
[21] Felsorolásuk a Függelékben
[22] A mezőérték kiolvasása *előtti* nullitásvizsgálatot nem támogatja minden adatbázis-kezelő rendszer, emiatt maradt ki a JDBC 1.2 API-ból.

```
  String s = r.getString("b");
  byte b[] = r.getBytes("c");
  System.out.println(i + " " + s + " " + b[0]);
}
stmt.close();
```

3.5. Hibakezelés

Ha az adatbázis kapcsolat során bármiféle hiba történik, Java szinten `SQLException` kivétel lép fel. A hiba szövegét az `SQLException.getMessage`, a kódját az `SQLException.getErrorCode`, az X/Open SQLstate konvenció szerinti állapotleírást az `SQLException.getSQLState` metódusok adják vissza.

3.6. Tranzakciókezelés

A `Connection` osztállyal reprezentált adatbázis-kapcsolatok *alapértelmezésben auto-commit módban* vannak. Ez azt jelenti, hogy minden SQL utasítás (`Statement`) egyedi tranzakcióként fut le és végrehajtása után azonnal véglegesítődik (commit). Az alapértelmezés átállítható a `Connection.setAutoCommit(false)` metódussal. Ebben az esetben a tranzakciót programból kell véglegesíteni illetve visszavonni a `Connection.commit` ill. `Connection.rollback` metódusokkal, a következő példának megfelelően:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
  "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
  "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

3.7. Adatbázis információk

Az adatbázissal kapcsolatos információkhoz (metaadatokhoz) a `DatabaseMetaData` interfészen keresztül lehet hozzáférni. Az interfészt megvalósító osztályt a kapcsolatot reprezentáló `Connection` példány `getMetaData` metódusa adja vissza:

```
DatabaseMetaData dbmd = conn.getMetaData();
```

A `DatabaseMetaData` interfész egyes metódusai a lekérdezett információtól függően egyszerű Java típussal, vagy `ResultSet`tel térnek vissza. Az alábbi táblázat néhány fontosabb metódust sorol fel.

Metódus elnevezése	**Visszatérési érték**	**Leírás**
`getDatabaseProductName`	`String`	Adatbázis termék elnevezése
`getDatabaseProductVersion`	`String`	Adatbázis termék verziószáma
`getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`	`ResultSet`	A megadott keresési feltételnek eleget tevő táblákat listázza

4. Az Oracle JDBC meghajtói

Az Oracle cég két kliensoldali JDBC meghajtót fejlesztett ki: a *JDBC OCI Drivert* és a *JDBC Thin Drivert*.

A **JDBC OCI Driver** a JDBC metódusokat OCI könyvtári hívásokként implementálja. Az Oracle Call Level Interface (OCI) az Oracle C nyelven megírt standard kliens oldali programozási felülete (adatbázis meghajtója), amely magasabb szintű fejlesztőeszközök számára biztosít hozzáférést az adatbázis-kezelő szolgáltatásaihoz. Egy JDBC-beli metódus (pl. utasításvégrehajtás) meghívásakor a JDBC OCI Driver továbbítja a hívást az OCI réteghez, amelyet az az SQL*Net ill. Net8 protokollon keresztül juttat el az adatbázis-kezelőhöz. A C könyvtári hívások miatt a JDBC OCI Driver platform- és operációs rendszer-függő; a natív kód miatt ugyanakkor hatékonyabb a tisztán Java-ban megírt Thin meghajtónál.

A **JDBC Thin Driver** teljes egészében Java-ban íródott meghajtó. A Thin Driver tartalmazza az SQL*Net/Net8 protokoll egyszerűsített, TCP/IP alapú implementációját; így egy JDBC metódushívást közvetlenül az adatbázis-kezelőhöz továbbít. A tiszta Java implementáció miatt a JDBC Thin Driver platformfüggetlen, így a Java applettel együtt letölthető. Az egyszerűsített implementáció miatt nem minden OCI funkciót biztosít (pl. titkosított kommunikáció, IP-n kívüli protokollok stb. nem támogatottak).

Az Oracle az adatbázisok címzésére – illeszkedve a JDBC konvencióhoz – a következő URL struktúrát használja:

```
jdbc:oracle:drivertype:user/password@host:port:sid
```

ahol a drivertype *oci7*, *oci8* vagy *thin* lehet; a host az adatbázis-szerver DNS-neve, a port a szerveroldali TNS listener portszáma, a sid pedig az adatbázis azonosítója. A felhasználói név és a jelszó a getConnection függvény második és harmadik paraméterében is megadható, ekkor az URL-ből a user/password szakasz elhagyandó.

5. Egy példa WebStart alkalmazásra

5.1. Java Web Start technológia

A WebStart a Java 1.4-től bevezetett, webről történő közvetlen alkalmazástelepítést és indítást könnyítő platformfüggetlen technológia. Egyetlen linkre való kattintással váltja ki a laikus felhasználók számára egyébként nehezen elsajátítható parancssori formulát. Az alkalmazás elindításán kívül továbbá biztosítja, hogy mindig a legfrissebb verzió legyen a kliens gyorsítótárába töltve, és az is fusson (az internetkapcsolat emiatt alap esetben kötelező). A WebStart használatához a Java alkalmazásunkon semmit nem kell változtatni, azt leszámítva, hogy kötelezően JAR csomag(ok)ba kell rendeznünk azt. Ezen kívül egy JNLP (Java Network Launching Protocol) telepítés-leíró állományt kell mellékelnünk és kézzel megszerkesztenünk.

A WebStarttal indított alkalmazás az appletekhez hasonlóan homokozóban (sandbox) fut, azaz olyan futtatókörnyezetben, mely korlátozza a helyi fájlrendszerekhez és a hálózathoz való hozzáférést. Az alkalmazás korlátlan jogokat kaphat azonban, ha minden komponense digitális aláírással rendelkezik, a JNLP-ben kimondottan kéri a plusz jogokat, és a felhasználó az aláíróban, illetve annak hitelesítés-szolgáltatójában megbízván ezeket explicit módon meg is adja (az engedélyezés csak első futáskor kell, később gyorstárazásra kerül). Számunkra ez azért fontos, mert egyébként nem tudnánk a kliensen futó programmal az adatbázishoz csatlakozni (hiszen olyan hálózati erőforrást szeretnénk használni, mely nem egyezik meg a letöltés helyével). A kliens egyéb erőforrásait a homokozóból a JNLP API rétegen keresztül tudjuk elérni (erre a mérésen nem lesz szükség).

5.2. Minta alkalmazás, JavaFX

A labor alkalmával a hallgatók rendelkezésére bocsátott minta alkalmazás – melynek forrása ZIP archívumban összecsomagolva letölthető a http://rapid.eik.bme.hu/~varsanyi.marton/jdbc/lab5jdbc.zip mérések során használt Oracle adatbázis szerverhez, majd lekérdezi és megjeleníti az 'Oktatas' sémában található 'Szemelyek' tábla első 20 rekordját.

Az alkalmazás könyvtárszerkezetének leírását, valamint fordításának és futtatásának menetét részletesen tartalmazza a tárgy JDBC mérésének weboldaláról letölthető hallgatói útmutató. Jelen leírás célja, hogy áttekintést adjon a minta alkalmazás funkcionalitásáról, architektúrájáról, valamint a létrehozása során alkalmazott technikákról. Jelen ismertetőben tehát az alkalmazás felületét, valamint Java forráskódját vesszük szemügyre, mely a fenti címről letölthető csomag *src*, illetve *resources* könyvtáraiban található fájlok tartalmát jelenti.

Az alkalmazás a labor témájának megfelelően Java programnyelven íródott és a JavaFX (https://en.wikipedia.org/wiki/JavaFX) névre hallgató GUI (Graphical User Interface) keretrendszert használja a felület létrehozásához. A JavaFX filozófiájának megfelelően a példa alkalmazás felépítése, architektúrája a klasszikus Model-View-Controller rétegződést követi. Ez kiegészül egy adatelérési réteggel, a Model rétegben található osztályok példányait ez a logikai réteg fogja szolgáltatni. Az *src* könyvtár az alábbi Java package-ket tartalmazza:

- **application** – Az alkalmazás Controller rétegét és a megjelenítéshez szükséges segéodsztályokat tartalmazza**.**
    - **AppMain.java** – Az alkalmazás belépési pontját tartalmazó osztály forráskódja; létrehozza és inicializálja a nézetet
    - **Controller.java** - Az alkalmazás vezérlő rétegét megvalósító osztály forráskódja; példányosítja az adatelérési réteget, feldolgozza a megjelenítési rétegtől érkező kéréseket, melyek kiszolgálásához az adatelérési réteg metódusait hívja, majd az eredményeket átadja a megjelenítési rétegnek.
    - **ComboBoxItem.java –** Egy példa arra, hogy hogyan tudunk tárolni adatokat a ComboBoxban**.**
- **dal** – Ebben a packageben vannak az adatelérési réteghez tartozó forrásfájlok.
    - **DataAccessLayer.java –** generikus interfész az adatelérési réteg szolgáltatásainak kiajánlására.
    - **ActionResult.java** – Az adatmódosító feladat lehetséges eredményeit tartalmazó enum.
- **dal.impl –** Ennek egyetlen – feladattípustól függő – forrásfájlja tartalmazza az adatelérési osztály szkeletonját.
- **dal.exceptions** - Ebben a package-ben vannak definiálva az adatelérési rétegben használt kivételek.

- **model –** Tartalmaz három osztályt, ami szintén feladattípusonként különböző, ezekkel hordozunk adatot a megjelenítés és az adatelérési réteg között. Található benne továbbá egy Person osztály, aminek segítségével valósul meg a példa SQL-lekérdezés eredményeinek megjelenítése.

A *resources* könyvtár a minta alkalmazásban egyetlen fájlt tartalmaz, mely a **View.fxml** nevet viseli. Ez a fájl írja le XML (Extensible Markup Language) (https://en.wikipedia.org/wiki/XML) nyelven az alkalmazás felületét, a felületen megjelenő vezérlőelemeket és azok kapcsolatát. JavaFX-specifikus fájl.

A minta alkalmazásban az adatbázishoz történő kapcsolódáshoz szükséges felhasználónév-jelszó páros az alkalmazás ablakának felső részén található beviteli mezőkben adható meg, a *Connect* gombra kattintva pedig létrehozza a kapcsolatot az adatbázissal. A kapcsolat státusza a *Connect* gomb mellett jelenik meg.

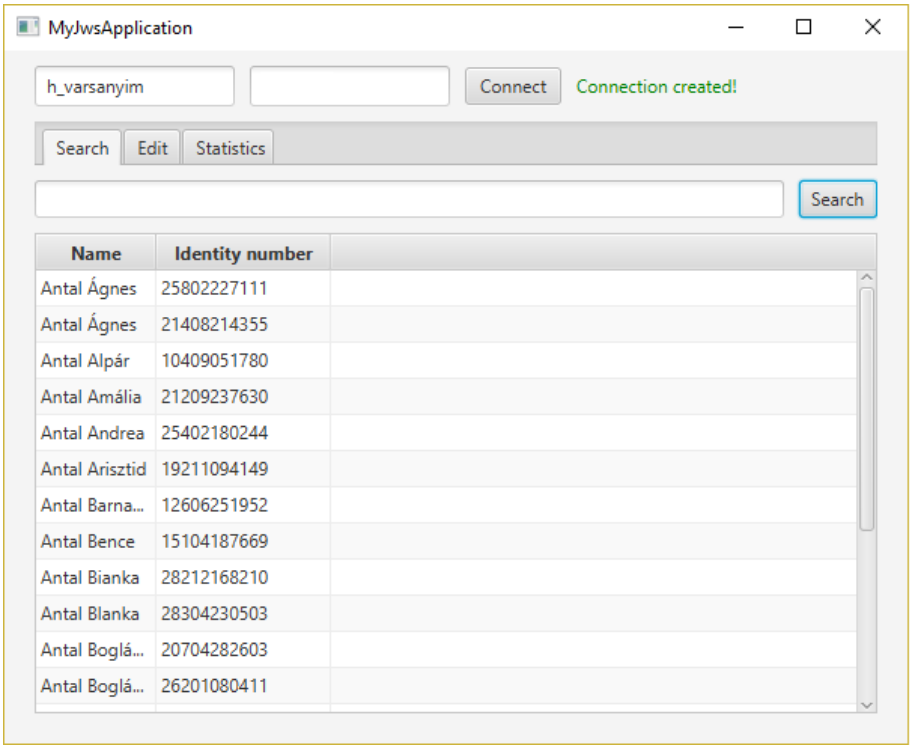
Az alkalmazás három fület tartalmaz, melyek rendre a következő nevet viselik: *Search*, *Edit* és *Statistics*. A fülek és a rajtuk elhelyezett vezérlőelemek szerepe kettős: egyrészről mintaként szolgálnak hasonló felületelemek létrehozásához, másrészt a labor során létrehozandó alkalmazás felületén ezekre az elemekre lesz szükség. A minta alkalmazás így részben meghatározza a gyakorlat során implementálandó alkalmazás felületét, másrészről segítséget is nyújt, hiszen így nagyobb figyelmet szentelhetünk a mérés lényegét jelentő adatbáziskezelés témakörének.

A minta alkalmazás *Search* névre hallgató fülén egy szövegbeviteli mező, egy gomb és egy táblázatelem kapott helyet. A *Search* gomb megnyomására a kezdetben üres táblázatba a már korábban említett lekérdezéshez tartozó rekordok jelennek meg.

Az *Edit* fülön példákat mutatunk címke (label), legördülő menü (dropdown list), szövegbeviteli mező és gomb elhelyezésére a felületen. A legördülő menüben két érték választható: ’Value A’ vagy ’Value B’.

Végül a *Statistics* fülön ismét egy gomb, illetve egy táblázat található, mely az utolsó mérési feladat megoldását hivatott segíteni.

Az alkalmazás felülete látható az alábbi ábrán sikeres csatlakozást követően:



Az alkalmazás forráskódjának tanulmányozása során első lépésként nézzük meg az alkalmazás belépési pontját – a *main* függvényt – is tartalmazó *AppMain.java* fájl tartalmát. A kód a szükséges JavaFX importokat követően az alkalmazás osztály létrehozásával kezdődik. JavaFX esetében ablakos alkalmazásunkat az *Application* osztályból származtathatjuk. Ahogy a kódból is látható, a főosztály neve *AppMain* lesz.

A fájl végefelé található a *main* függvény, mely paraméterként az esetleges parancssori argumentumokat kaphatja. A függvény törzse egyetlen sort tartalmaz, mely példányosítja és elindítja alkalmazásunkat.

Az alkalmazás indulása során elvégzendő tevékenységeket a *start* nevű metódusban kell implementálni, míg a *stop* metódus az alkalmazás leállítása előtt hajtódik végre. A *start* függvény elején az *FXMLLoader* objektum segítségével betöltjük az alkalmazás felületét leíró XML fájlt, nevezetesen a *resources* könyvtárban található *View.fxml* állományt. Ekkor létrejön az alkalmazás felületét képező vezérlőelemek hierarchiája (az XML fájl alapján), melynek gyökérelemére referenciát is kapunk a *viewRoot* változóval.

A JavaFX alkalmazások felületének létrehozásánál a legfelsőbb szintű konténert a *Stage* objektum jelenti. Ezen objektum gyerekeként kell beállítani azt a *Scene* objektumot, ami az ablakban ténylegesen megjelenő elemek konténere. Végül a *Scene* objektum gyerekeként a korábbi *viewRoot* objektumot adhatjuk meg, mely az XML fájlban leírt legfelső szintű felületelem.

A fenti beállítások elvégzését követően a *setTitle* metódushívással megadjuk az ablak nevét, végül megjelenítjük azt. Az előbbi lépések láthatók az alábbi kódrészletben:

```
// Create a loader object and load View and Controller
final FXMLLoader loader = new
FXMLLoader(getClass().getClassLoader().getResource("resources/View.fxml"));
final VBox viewRoot = (VBox) loader.load();
```

```
// Get controller object and initialize it
controller = loader.getController();

// Set scene (and the title of the window) and display it
Scene scene = new Scene(viewRoot);
primaryStage.setScene(scene);
primaryStage.setTitle("MyJwsApplication");
primaryStage.show();
```

A fenti kódrészletben még az is látható, hogy elkérjük az FXML fájl alapján létrehozott felület vezérlő osztályának referenciáját, mely a nézetet hivatott kezelni.

Következő lépésként vizsgáljuk meg az alkalmazás felületét, annak elrendezését, megjelenését leíró FXML fájlt. Az XML fájlok, így esetünkben az FXML fájl is (*View.fxml*) a HTML fájlokhoz hasonlóan ún. tag-ekből épülnek fel. A tageket csúcsos zárójelek közé írjuk, mint azt láthatjuk az alábbi példa esetében is:

```
<VBox fx:controller="application.Controller"
xmlns:fx="http://javafx.com/fxml/1"
      fx:id="rootLayout" alignment="CENTER" spacing="10" prefWidth="600"
      prefHeight="460" minWidth="600" minHeight="460">
```

FXML fájlokban ezek a tagek egy-egy felületelemet írnak le, mint pl. egy gombot, címkét, szövegbeviteli mezőt, vagy egy elrendezést (layout). A tagek (valójában párok) rendelkeznek egy nyitó és egy záró taggel.

Valamennyi tag egyedi névvel rendelkezik, melyet a '<' jel után írhatunk. A fenti példa esetében ez a *VBox*-nak felel meg. Ez jelenti az adott felületelem megnevezését. Ezt követően soroljuk fel a tag ún. attribútumait, vagyis azokat a paramétereket, melyek a felületelem egyes beállításainak értékeit adják meg. Valamennyi beállítást egy-egy attribútumnév jelöl, mint pl. a *prefHeight, prefWidth* attribútumok a fenti kódrészletben, melyek jelen esetben a *VBox* elem preferált magasságát és szélességét határozzák meg. Az attribútumokhoz értéket *név=érték* formában rendelhetünk, ahogy a példában is látható. Az értékek minden esetben idézőjelek közé írandók.

Az importokon és az alkalmazás elején található speciális *xml* tagen kívül a tageknek van egy záró párjuk is. Egy taget a *</tag_neve>* formában zárhatunk le. Egy tag nyitó és záró tagjén belül további taget/tageket helyezhetünk el, melynek jelentése, hogy a tag által reprezentált felületelembe további felületetelemet ágyazunk. Ez elsősorban a konténer típusú elemek esetén lényeges, de természetesen vannak más esetek is. Amennyiben egy felületelembe nem ágyazunk további elemeket, mint pl. egy gomb esetén, úgy logikusan a nyitó és záró tagek közötti rész üresen marad. A jobb áttekinthetőség érdekében ezt rövidíthetjük úgy, hogy a nyitó és záró taget összevonjuk a következő formában: *<tag_neve esetleges_attribútumok />*.

Az XML fájlok – és így az FXML fájlunk is – kötelezően egy *xml* taggel kezdődik, mely esetünkben az alábbi:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Ebben megadjuk az XML verzióját és a kódolást, mely UTF-8 lesz. Ezt követően szükséges importálni (természetesen az XML formátumnak megfelelően) a felület leírása során

felhasznált elemek osztályát. Erre mutat példát a következő sor, melyben a gomb felületelem (*Button*) osztályát importáljuk:

```
<?import javafx.scene.control.Button?>
```

A felület részeként számos elem megadható, ezek tárházából csak kisebb ízelítőt próbál nyújtani a minta alkalmazás. Az FXML fájlban láthatunk példát gombok, címkék, szövegmezők, fülek, legördülő menü, táblázat stb. elhelyezésére, illetve kapunk példát elrendezések, ún. layout-ok használatára is.

Ilyen layout a korábbi példában szereplő *VBox* felületelem is. Ez egy olyan konténer, mely egymáshoz képest vertikálisan helyezi el a belé ágyazott elemeket. A *VBox*-os példa attribútumaiból továbbá az is látszik, hogy a layout-ban elhelyezett elemek egymástól fixen 10 pixel távolságra kerülnek (lsd.: *spacing* attribútum értéke), valamint, hogy a beágyazott elemek középre igazítva jelennek meg (*alignment* attribútum). Érdemes megnéznünk az FXML fájlban azt is, hogy amennyiben szeretnénk valamekkora méretű üres területet beállítani a konténer elem kerete és a tartalom között, úgy azt a *padding* tag segítségével tehetjük meg, ahol a *padding* tagbe ágyazott *Insets* nevű tag attribútumaiban kell megadnunk a keret (border) alsó, felső, jobb és a bal oldali részénél beállítandó szabad terület mértékét.

A *VBox*-hoz hasonló a *HBox* névre hallgató layout, csak míg előbbi esetben vertikálisan, addig utóbbi esetben horizontálisan kerülnek elhelyezésre a layout-ba ágyazott elemek.

Fontos kiemelni négy speciális attribútumra, melyek közül háromra a *VBox* esetén példa is látható. Az első az *fx:controller* attribútum, mely megadja a felület kezelését, az eseménykezelést ellátó osztály nevét. Ahogy a példában látható, ezt a szerepet – a korábban írtakkal összhangban – a *Controller* nevű osztály tölti be. Ezt az attribútumot egyetlen alkalommal lehet csak megadni az XML alapú felület gyökérelemének tagjében (a mi esetünken a legkülső *VBox* tagben). Az attribútum értékeként azért írtunk *application.Controller*t és nem simán *Controller*t, mert a minta alkalmazás valamennyi osztálya egy *application* névre hallgató csomagon (package) belül található. (Ennek a package-nek a definiálása megtalálható valamennyi Java fájl elején.)

Az *xmlns:fx* attribútum azt a névteret definiálja, melyből a használt tagek nevei származnak. A névtér jelen esetben a Java *fxml* névtér, melynek pontos hivatkozása a következő: *http://javafx.com/fxml/1*. Névteret is csak egy alkalommal adunk meg, a gyökérelem tagjében.

Az *fx:id* attribútum a felületelem egyedi azonosítóját adja meg. Ez az azonosító azért különösen fontos a számunkra, mert ezen az ID-n keresztül tudunk majd a felület kezelését ellátó osztályból hivatkozni a felületelemre.

Végezetül amennyiben eseményeket is szeretnénk kezelni, úgy valamilyen módon eseménykezelőket (eseménykezelő metódusokat) is kell rendelnünk az egyes vezérlőelemekhez. Ennek módja az, hogy a vezérlőelem (pl.: *Button*) *onAction* nevű attribútumában megadjuk az eseménykezelő metódus nevét egy ’#’ karakterrel az elején. Erre láthatunk példát alább:

```
<!-- Search button -->
<Button fx:id="searchButton" text="Search" onAction="#searchEventHandler"
/>
```

A minta alapján a *searchButton* azonosítójú gomb megnyomásának hatására a nézetet kezelő osztályban található *searchEventHandler* metódus hívódik meg, mely lekezeli az eseményt. A példából az is látszik továbbá, hogy megjegyzéseket is elhelyezhetünk az XML kódban, melyeket a <!-- nyitó és --> záró részek közé kell írnunk.

Az XML fájl leírásának végén megjegyezzük, hogy az XML nyelvvel a hallgatók az 5. (XSQL című) mérés során találkozhatnak újra, melynek keretében mélyrehatóbban ismerkedhetnek azzal.

Leírásunk folytatásaként térjünk át a nézet kezelését megvalósító *Controller* osztályra, mely a *Controller.java* fájlban található. Ennek az osztálynak a konstruktorában példányosítjuk az adatelérési réteget, a következőképpen (a példában a VIDEO feladatsorhoz tartozó osztály szerepel):

```
public Controller() {
	dal = new VideoDal();

}
```

A továbbiakban a *Controller* osztály ennek az adatelérési rétegnek a segítségével fog kapcsolódni az adatbázishoz, és a lekérdező-, illetve adatmódosító műveleteket elvégezni.

Az osztályon belül *@FXML* dekorátorral kell ellátnunk azokat a metódusokat, melyeket hivatkozunk az FXML fájlból. Ilyenek lesznek az eseménykezelő metódusok, mint pl. a *Connect* vagy a *Search* gomb eseménykezelő metódusai. Eseménykezelő metódusra mutat példát az alábbi kódrészlet:

```
@FXML
public void searchEventHandler() {
	//TODO: replace this query with your solution.
	try {
		List<Person> people = dal.sampleQuery();
		searchTable.setItems(FXCollections.observableArrayList(people));
	} catch (NotConnectedException e) {
		e.printStackTrace();
	}

}
```

Az eseménykezelő metódusokban általában el is akarunk érni adott felületelemet, melyen módosítást hajtunk végre (pl. módosítjuk egy szövegbeviteli mező tartalmát). Ehhez az szükségeltetik, hogy rendelkezzünk valamilyen referenciával az érintett felületelemre. A felületen megjelenő egyes vezérlőelemekre, layout-okra úgy tudunk hivatkozni Java kódból, hogy a kód elején létrehozunk a felületelemek ID-jával azonos nevű objektumokat, melyek típusa (osztálya) megegyezik a felületelemek típusával (nevével). Ezen kívül az objektum deklarálása felett el kell helyeznünk a már megimert *@FXML* dekorátort. Ehhez példaként szolgál a következő kódsor:

```
@FXML
private TextField usernameField;
```

A mintában a *usernameField* ID-jú szövegmező elemhez hozunk létre objektumot *TextField* osztály megadása mellett. Természetesen a szükséges JavaFX osztályokat (mint pl. a *TextField* is) importálnunk kell a kód elején.

Fontos még kiemelnünk az *initialize* metódust, mely automatikusan meghívódik a felület felépítését követően, így a felületelemek inicializálását (pl.: szövegmezők törlését) ezen függvényen belül muszáj elvégeznünk. Ennek az az oka, hogy az osztály példányosításakor a felületelemek még NULL értékűek. Az *initalize* metódus lefutásakor azonban már biztosak lehetünk abban, a keretrendszer már hozzákötötte a megfelelő vezérlőt a tagváltozóhoz. Ebben a metódusban érdemes feltöltenünk a ComboBoxokat a megfelelő értékekkel, illetve a táblázatok oszlopaiban az adatkötést elvégeznünk:

```
@Override
public void initialize(URL location, ResourceBundle resources) {
	// TODO: initalize property-value factories
	//EXAMPLE: sampleCombo stores two strings.
	sampleCombo.getItems().add(new ComboBoxItem<String>("Value A", "a"));
	sampleCombo.getItems().add(new ComboBoxItem<String>("Value B", "b"));

	//EXAMPLE: this is how we bind the private variables to a data cell
	nameColumn.setCellValueFactory(new PropertyValueFactory<>("name"));
	identityNumberColumn.setCellValueFactory(
		new PropertyValueFactory<>("identityNumber"));

}
```

Az adatkötés során az történik valójában, hogy megadjuk az adott sorban lévő objektummelyik tagváltozójából olvassa ki a keretrendszer a megjelenítendő értéket. Ez a név esetében *name* tagváltozó, a személyi igazolványszám esetében pedig az *identityNumber*.

Végezetül az adatelérési réteg (**Dal.java, pl: VideoDal.java*) a fejezet elején leírtaknak megfelelően tartalmazza az adatbáziskezeléssel kapcsolatos logikát. Ebben az osztályban történik az adatbázishoz való csatlakozás, illetve a példaként bemutatott lekérdezés megvalósítása. Az adatbázishoz való csatlakozás előtt betöltjük az Oracle JDBC meghajtóját. Ez látható a következő kódrészletben:

```
// Load the specified database driver
Class.forName(driverName);
```

Az adatbázisból adatok lekérdezésére egy példát a *sampleQuery* metódusban láthatunk:

```
@Override
public List<Person> sampleQuery() throws NotConnectedException {
	checkConnected();
	List<Person> result = new ArrayList<>();
	try (Statement stmt = connection.createStatement()) {
		try (ResultSet rset = stmt.executeQuery(
				"SELECT nev, szemelyi_szam FROM OKTATAS.SZEMELYEK "
				+ "ORDER BY NEV "
				+ "OFFSET 0 ROWS FETCH NEXT 20 ROWS ONLY")) {
			while (rset.next()) {
				Person p = new Person(rset.getString("nev"),
						rset.getString("szemelyi_szam"));
				result.add(p);
			}
			return result;
		}
	} catch (SQLException e) {
		e.printStackTrace();
		return null;
```

```
        }

}
```

Az első sorban ellenőrizzük, hogy kapcsolódva vagyunk-e már az adatbázishoz, egy privát metódus segítségével. Létrehozunk egy *Statement* típusú objektumot, majd lefuttatjuk a lekérdezést. Ennek rekordjain végigiterálunk, és mindegyikből létrehozunk egy *Person* objektumot, amit az eredményként visszaadott listába tesszük bele. A labor során ajánlott – a mintakódnak megfelelően – a Java 8-as try-with-resources szerkezet használata. Lásd bővebben: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. Az adatbázistól érkező hibákat a kivételkezelő ágakban SQLException-ök formájában kaphatjuk el.

A model packageben lévő osztályok egyszerű POJO (Plain old Java Object) osztályok, a privát tagváltozókat getter-setter metódusokkal érhetjük el.

A fentiekben vázolt JavaFX-es alkalmazás WebStart-os „keretet” kap azáltal, hogy létrehozunk egy WebStart-ot vezérlő konfigurációs JNLP állományt:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="http://rapid.eik.bme.hu/~xxxxxx/jdbc"
href="application.jnlp">
      <information>
            <title>My Java Webstart JDBC Application</title>
            <vendor>Test student</vendor>
            <icon href="logo.png" kind="default" />
      </information>
      <security>
            <all-permissions />
      </security>
      <resources arch="" os="">
            <j2se version="1.7+" />
            <jar href="ojdbc7.jar" />
            <jar href="MySignedApplication.jar" main="true" />
      </resources>
      <application-desc />
</jnlp>
```

A WebStart-ot beágyazó HTML oldal részlete:

```
<a href="application.jnlp">
      <font size="4">My Java Webstart JDBC Application</font><br />
      <img src="logo.png" alt="My Java Webstart JDBC Application" />
</a>
```

6. Felhasznált irodalom

1. *The JDBC API Verson 1.20*, Sun Microsystems Inc.
2. S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner: *JDBC 2.0 API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*
3. S. Kahn: *Accessing Oracle from Java*, Oracle Co.
4. Nyékiné et al. (szerk.)*: Java 1.1 útikalauz programozóknak*, ELTE TTK Hallgatói Alapítvány

7. Függelék: Oracle adattípusok elérése JDBC-ből

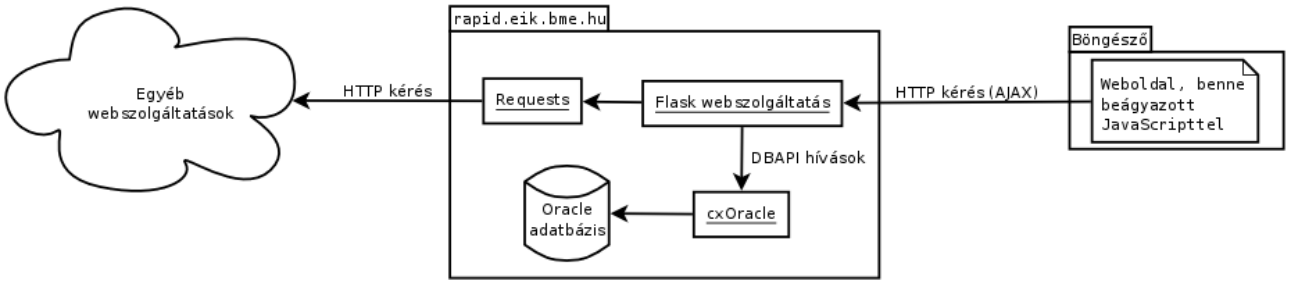
Java típus	Lekérdező metódus	CHAR	VARCHAR2	NUMBER	DATE
byte	getByte	●	●	●	○
short	getShort	●	●	●	○
int	getInt	●	●	●	○
long	getLong	●	●	●	○
float	getFloat	●	●	●	○
double	getDouble	●	●	●	○
java.Math.BigDecimal	getBigDecimal	●	●	●	○
boolean	getBoolean	●	●	●	○
String	getString	●	●	●	●
java.sql.Date	getDate	○	○	○	●
java.sql.Time	getTime	○	○	○	●
java.sql.TimeStamp	getTimeStamp	○	○	○	●

Jelölések:
○: a getXXX metódus *nem használható* az adott SQL típus elérésére
●: a getXXX metódus *használható* az adott SQL típus elérésére
●: a getXXX metódus *ajánlott* az adott SQL típus elérésére

IV. labor: SOA szemléletű rendszer REST architektúrában

Szerző: Veres-Szentkirályi András

A SOA labor során egy, a SOA elvei mentén működő rendszert fogunk készíteni, szerveroldalon Python, böngészőoldalon JavaScript használatával. A segédletben először a SOA alapjai kerülnek bemutatásra, majd a szerveroldalon futó Python SOA szerver és kliens implementációk, végül a böngészőben futó megoldások. Az alábbi áttekintő ábrán látható az egyes felhasznált komponensek egymáshoz való viszonya, a segédlet olvasása során érdemes időről-időre visszagörgetni ide.



1. SOA alapok

1.1. Bevezetés és történelem: SOA és webszolgáltatások

A SOA (szolgáltatás-orientált architektúra) természetes evolúciós lépés az informatikai rendszerekben. A strukturált, objektum-orientált, komponens-orientált programozásra építve szolgáltatás-orientált rendszerek esetén a kód újrafelhasználása a rendszer szolgáltatásokra bontásán alapul. Ezek jellemzője, hogy elérésük szabványos interfészen keresztül történik, így szemben például az objektum-orientált rendszerekkel, a szolgáltatás és igénybevevője akár eltérő programnyelven is íródhat, eltérő operációs rendszeren vagy számítógépen is futhat. A fent vázolt evolúció vonalát folytatva így csökken a komponensek közti függőség és csatolás, ennek viszont alapvető feltétele a már említett szabványos felületek és formátumok használata. Tipikusan ilyen az XSQL mérésen majd mélyebben bemutatott XML, illetve az ezen a mérésen bemutatásra kerülő, egyre jobban terjedő JSON.
A webszolgáltatások a SOA egyik legnépszerűbb implementációit képviselik, nevüket a HTTP protokoll használatáról kapták. Mint nyílt, egyszerű és elterjedt szabvány, megfelel a SOA rá szabott feltételeinek, és a bináris protokollokkal (OMG/CORBA, Microsoft/DCOM) szemben egyszerűbben feldolgozható és könnyebben továbbítható tűzfallal izolált hálózati szegmensek között is, ezáltal biztonsági szempontból is kedvezőbb. Korábban egyeduralkodó volt és most is elterjedt protokoll az 1998-ban a Microsoft berkeiben megalkotott SOAP, ami XML-t használt a kérések és válaszok leírására, de legtöbbször szintén a HTTP protokoll felett került megvalósításra. Ennek szemantikája leginkább távoli metódushívásnak feleltethető meg, a kérésben a metódus neve és paraméterei kerültek leírásra, majd a válaszban a metódus visszatérési értéke került visszaküldésre.

1.2. A laboron bemutatott megvalósítás: REST

A REST (Representational State Transfer) egy megvalósítása a SOA-nak, inkább megvalósítási stílusként, vezérelvek gyűjteményeként fogható fel, mint protokollként. A SOAP-pal szemben a REST a HTTP eredeti, webes szemantikáját terjeszti ki, az elérhető távoli erőforrások állapotát HTTP kéréssel lehet lekérdezni és/vagy módosítani. Szabvány ugyan nem határozza meg a használt átviteli formátumot, elterjedten használt megoldás az XML és JSON. A szolgáltatás elérésének paraméterei tipikusan a lekért URL-ben kerülnek átadásra, opcionálisan a HTTP kérés fejléce (HTTP fejlécek) is használhatók a célra. Az erőforrások tipikusan az adatbázis tábláival (entitáshalmazokkal) állíthatók párhuzamba, a HTTP verbek (lásd lejjebb) pedig a következő SQL utasításokkal.

Verb	Útvonal	SQL utasítás
GET	/hallgatok.xml	SELECT * FROM hallgatok
GET	/hallgatok/42.xml	SELECT * FROM hallgatok WHERE id = 42
POST	/hallgatok.xml	INSERT INTO hallgatok (...) VALUES (...)
PUT	/hallgatok/42.xml	UPDATE hallgatok SET ... WHERE id = 42
DELETE	/hallgatok/42.xml	DELETE FROM hallgatok WHERE id = 42

Gyakorlatban POST és PUT esetén a kérés törzse tartalmazza a beszúrandó/frissítendő adatokat, GET és DELETE esetén a lekérdezni/törölni kívánt entitás azonosítói az URL-ben kerülnek megadásra.
Az első oszlopban található HTTP verb magyarul leginkább metódusnak nevezhető, a HTTP protokollon közlekedő kérés típusát jelzi. A verb kifejezést használjuk a továbbiakban, hogy megkülönböztessük az objektum-orientált paradigma metódus kifejezésétől. Bár HTTP verb elvileg tetszőleges string lehet, a gyakorlatban a fentiek használata ajánlott, mivel szemantikájuk bejáratott és általánosan elfogadott. Például mivel a GET kérésnek nem szabadna megváltoztatnia az erőforrás állapotát, bármikor újraküldhető, ill. közbenső (pl.

proxy-) szerverek és szolgáltatások gyorsítótárban tárolhatják a választ a szolgáltatás és a hálózat terhelésének mérséklése érdekében.
A weben több lista is található különféle, bárki számára elérhető REST webszolgáltatásokról, ilyen például a ProgrammableWeb, ahol a segédletben bemutatott szolgáltatások is elérhetők.

1.3. HTTP protokoll és a REST kapcsolata

Egy REST architektúrájú webszolgáltatás igyekszik a HTTP protokoll beépített lehetőségeit kihasználni metaadatok átadására – szemben például a SOAP-pal, amely ezt a HTTP felett oldja meg. A labor során használt metaadatok kiterjednek a szolgáltatás-hívás sikerességére, az üzenet (kérés és válasz) tartalmának típusára, és autentikációra is, ebben a pontban az elsővel foglalkozunk részletesen.
Egy tipikus HTTP/1.1 válasz egy ún. státusz sorral kezdődik, amely tartalmazza az alkalmazott HTTP protokoll verziót, és a HTTP kérés feldolgozásának állapotát gépi és emberi feldolgozásra alkalmas változatban is. Előbbi egy háromjegyű szám, tipikusan ez kerül értelmezésre a kliensek által, és mivel hierarchikus felépítésű, mindjárt az első számjegy elárul néhány dolgot a tranzakció állapotáról.

- **1xx** – informális, a kérés feldolgozása még nem történt meg
- **2xx** – sikeres feldolgozás, a kérést elfogadta a kiszolgáló
- **3xx** – átirányítás, a kliens feladata további lépések megtétele a siker érdekében
- **4xx** – kliens hiba, a kérés nem megfelelő valamilyen oknál fogva
- **5xx** – szerver hiba, a kérést képtelen kiszolgálni valamilyen oknál fogva

A kódokat a 2616-os RFC 10. szekciójában szabványosították, a leggyakoribb kód a 200 (OK), melyet a legtöbb szerveroldali megoldás alapértelmezetten küld. Ezen kívül még az alábbi kódok ismerete javasolt a laborfeladatok megoldásához. (A kódot zárójelben követi a szabvány szerinti szöveges, ám kizárólag emberi felhasználásra szánt változat.)

- **201** (Created) – POST kérés esetén a sikeres létrehozási műveletet jelezheti
- **204** (No Content) – PUT és DELETE kérés esetén a sikeres módosítási/törlési műveletet jelezheti, mivel nincs szükség további információ átadására a válasz törzsében (szemben a 201-gyel, ahol például az új elem egyedi kulcsa kerülhet visszaadásra)
- **404** (Not Found) – talán a legismertebb státusz kód, azt jelzi, hogy a kért erőforrás nem létezik, például `GET /hallgatok/42.xml` jellegű kéréseknél mindenképpen javasolt, ha a 42-es egyedi kulccsal nem létezik rekord; ha azonban a kérésre több rekord is visszaadásra kerülhet (pl. kereső vagy listázó végpont), akkor szemantikailag megfelelő egy 200-as válasz is, üres jelentésű törzzsel (pl. JSON formátum esetén `{"results": []}` ilyen lehet).

1.4. A formátum: JSON

A JSON egy olyan pehelysúlyú adatátviteli formátum, mely a JavaScript nyelv szabvány szerint korlátozott részhalmaza, így minden JSON szabványnak megfelelő karaktersorozat egyben érvényes JavaScript kifejezés is. Hasonló célokra alkalmazható, mint az XML, viszont annál jóval egyszerűbb, így feldolgozása kevesebb kóddal és hibalehetőséggel megoldható. Érvényes JSON kifejezések a következők:

- Unicode stringek idézőjelek közé zárva, különleges karaktereket (ékezetes betűk, sortörés, idézőjel, stb.) escape-elve: `"alma"`, `"M\u0171egyetem"`
- Egész számok: `42`, `-5`
- Lebegőpontos tizedestörtek (tizedes pont használatával): `5.5`, `-273.1`
- Null érték: `null`
- Logikai értékek: `true`, `false`

- Érvényes JSON kifejezésekből alkotott lista: `["alma", 42, null]`, `[]`, `[1.1]`
- Stringek és érvényes JSON kifejezések párjaiból alkotott szótár: `{"telefon": "phone"}`, `{"1": "Jan", "2": "Feb"}`

A *whitespace* karaktereket (újsor, tabulátor, szóköz) a feldolgozók figyelmen kívül hagyják, így ha a maximális tömörség a cél, egy JSON kimenet akár egyetlen hosszú sor is lehet, míg ha az olvashatóság is fontos, a legtöbb könyvtár képes tagolt, behúzott (indentált) kimenet előállítására. A következő két kimenet például ekvivalens:

```
{"targynev": "Szoftver Labor 5", "laborok": ["Oracle", "SQL", "JDBC", "SOA", "XSQL"]}
```

```
{
  "targynev": "Szoftver Labor 5",
  "laborok": [
    "Oracle",
    "SQL",
    "JDBC",
    "SOA",
    "XSQL"
  ]
}
```

Látható, hogy az XML-hez képest nincsenek névterek és a karakterkódolás sincs megadva, mivel a kötelező string kódolás miatt egy szabályos JSON kimenet kizárólag 0-127 közötti bájtokat tartalmazhat, melyeket ASCII szerint kell értelmezni. Ennek megfelelően ASCII-vel kevés átfedést mutató (pl. orosz, kínai, japán) nyelven írt stringek esetében a kimenet mérete többszöröse lehet egy azonos jelentéstartalmú XML-ének – ez az ára az egyszerű szabványnak és feldolgozó szoftvereknek. Megjegyzendő azonban, hogy JSON legtöbbször HTTP felett kerül továbbításra, mely tartalmaz lehetőséget – többek között – *gzip* tömörítésre, így a legtöbb esetben a JSON által okozott méretkülönbség elhanyagolható.

1.5. Teszteléshez svájci bicska: cURL

A *cURL* egy parancssoros kliens többek között HTTP protokollhoz, így webszolgáltatások tesztelésére is használható. A legtöbb UNIX-szerű rendszeren (például a rapidon is) telepítve van, de honlapjáról letölthető szinte minden operációs rendszerre, még kevésbé elterjedtekre is. A programot a `-h` (help) kapcsolóval elindítva részletes súgót kapunk, számtalan paraméterének kimerítő bemutatása nem célja e segédletnek, így alább a REST tesztelés szempontjából releváns use-case-ek kerülnek bemutatásra. GET kérést indítani a legegyszerűbb, ekkor az egyetlen paraméter a lekérni kívánt URL.

```
$ curl http://currencies.apps.grandtrunk.net/getlatest/eur/huf
298.880251501
```

Amennyiben a hibakereséshez szükséges, a `-v` (verbose) kapcsolóval részletesebb kimenet kérhető.

```
$ curl -v http://currencies.apps.grandtrunk.net/getlatest/eur/huf
* About to connect() to currencies.apps.grandtrunk.net port 80
*   Trying 66.33.208.161... connected
* Connected to currencies.apps.grandtrunk.net (66.33.208.161) port 80
> GET /getlatest/eur/huf HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5 OpenSSL/0.9.8b
zlib/1.2.3 libidn/0.6.5
> Host: currencies.apps.grandtrunk.net
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< Date: Sun, 05 Apr 2015 09:11:39 GMT
< Server: Apache
< Cache-Control: public
< Expires: Mon Apr  6 04:11:39 2015
< X-Powered-By: Phusion Passenger 4.0.59
< Content-Length: 13
< Status: 200 OK
< Vary: Accept-Encoding
< Content-Type: text/plain; charset=utf-8
Connection #0 to host currencies.apps.grandtrunk.net left intact
* Closing connection #0
298.880251501
```

Amennyiben adatot kell beküldeni a szolgáltatásnak, a következő kapcsolók lehetnek segítségünkre. Amennyiben egy kapcsoló paramétere szóközt tartalmaz, mindenképp idézőjelek közé kell tenni, egyébként elhagyható. Amennyiben idézőjelek is szerepelnek a tartalomban (ami JSON esetében nem ritka), aposztróf karakter is használható idézőjel helyett a paraméter határolására.

- `--request <verb>` a megadott HTTP verbet használja `GET` helyett
- `--header "Fejlec: ertek"` hozzáfűzi a megadott fejlécet a HTTP kéréshez, leginkább `Content-Type` megadására szokás REST esetén használni, természetesen többször is megadható
- `--data "foo bar qux"` a megadott adatot továbbítja a kérés törzsében, változtatás nélkül

Alább látható egy példa egy POST kérésre, mely a GitHub Gist szolgáltatásában hoz létre egy `SOA labor` tartalmú `szoftlab.txt` nevű fájlt.

```
$ curl --request POST --header "Content-Type: application/json" --data '{"public":
    true,     "files":     {"szoftlab.txt":     {"content":     "SOA     labor"}}}'
    https://api.github.com/gists
{
  "url": "https://api.github.com/gists/5345409",
  "forks_url": "https://api.github.com/gists/5345409/forks",
  "commits_url": "https://api.github.com/gists/5345409/commits",
  "id": "5345409",
  "git_pull_url": "https://gist.github.com/5345409.git",
  "git_push_url": "https://gist.github.com/5345409.git",
  "html_url": "https://gist.github.com/5345409",
  "files": {
    "szoftlab.txt": {
      "filename": "szoftlab.txt",
      "type": "text/plain",
      "language": null,
      "raw_url":
    "https://gist.github.com/raw/5345409/4008674e9bc47bdf54a28260e7a502f30aca834d/
    szoftlab.txt",
      "size": 9,
      "content": "SOA labor"
    }
  },
  "public": true,
  "created_at": "2013-04-09T12:43:11Z",
  "updated_at": "2013-04-09T12:43:11Z",
  "description": null,
  "comments": 0,
```

```
  "user": null,
  "comments_url": "https://api.github.com/gists/5345409/comments",
  "forks": [

  ],
  "history": [
    {
      "user": null,
      "version": "05aa55498c23f46abe542c61ec6fd4e4d75910ea",
      "committed_at": "2013-04-09T12:43:11Z",
      "change_status": {
        "total": 1,
        "additions": 1,
        "deletions": 0
      },
      "url":
    "https://api.github.com/gists/5345409/05aa55498c23f46abe542c61ec6fd4e4d75910ea
    "
    }
  ]
}
```

2. Szerveroldal: Python

2.1. Programnyelv: Python

A Python egy olyan programozási nyelv, mely lehetővé teszi – akár egy programon belül is – több (funkcionális, procedurális, objektum-orientált) paradigma használatát is. E tulajdonsága miatt többek között rövid scriptek nyelveként terjedt el, azonban mára a helyzet megváltozott, és jelenleg egyike azoknak a nyelveknek, melyek jól használhatók webszolgáltatások és -alkalmazások készítésére. Jelenleg a Python 2.x verziói a legelterjedtebbek, így a labor során tett megállapítások erre lesznek igazak.

Kezdők számára legszembetűnőbb tulajdonsága, hogy a blokkok egymáshoz való viszonyát nem kezdő és lezáró karaktersorozatok, hanem a behúzás mértéke definiálja. A behúzás történhet tabulátor vagy szóköz karakterrel és ezek kombinációjával is, a hivatalos Python dokumentáció blokkonként 4 szóközt ajánl. Érdemes Python kód szerkesztéséhez olyan szövegszerkesztőt választani, amelyben beállítható a behúzás megjelenítése vagy automatikus kezelése, különben érdekes hibajelenségekkel találkozhatunk, amelyek forrása nehezen ismerhető fel.

A nyelv erősen típusos, azaz például PHP-tól eltérően sosem történik automatikus típuskonverzió, azonban a változókat és típusukat nem szükséges előre deklarálni, a változók első értékadásukkor jönnek létre. A következő kódrészlet végére például `b` lehet `int` vagy `str` típusú is.

```
if a > 5:
    b = 10
else:
    b = "kisebb"

print b
```

Az alapvető típusok a következők, a példákban látható, hogyan adható meg kódban egy adott típusú konstans érték:

- `int`: egész számok, pl. `42`
- `float`: lebegőpontos számok, pl. `5.5`
- `bool`: logikai érték, pl. `True`, `False`

- `NoneType`: null érték, `None`
- `str`: bájtokból álló string, pl. `"szoftlab5"`
- `unicode`: karakterekből álló Unicode string, pl. `u"M\u0171egyetemi hallgató"`
- `list`: tömb, pl. `[42, 5.5, "szoftlab5"]`
- `tuple`: „ennes”, pl. `(42, 5.5, "szoftlab5")`, `("SOA",)`
- `dict`: szótár, pl. `{"telefon": "phone"}`, `{1: "Jan", 2: "Feb"}`

Látható, hogy az alapvető típusok reprezentációja nagyban hasonlít a JSON-höz, egy különbség a stringek kezelése (ebből a szempontból fejlettebb a Python 3, illetve itt nem kötelező, de megengedett az ASCII-n kívüli karakterek escape-elése). Fontos, hogy a Java és .NET string típusaihoz hasonlóan **a `str` és `unicode` példányok „rögzítettek”**, így az ezeken végzett módosító műveletek (`lower`, `upper`, `replace`, stb.) nem a metódushívás „alanyát” módosítják, hanem **a módosított stringet visszatérési értékben kapjuk meg**.
Szintén különbség a `tuple` típus megléte, mely hasonlít a tömbökre, fontos különbség viszont, hogy stringekhez hasonlóan „rögzített”, azaz létrejötte után sem az elemek száma nem módosítható, sem az elemek nem cserélhetők más értékre. Ennek megfelelően több elem visszatérési értékként való átadásánál szokás alkalmazni, pl. adatbázis sor lekérdezése. Egyébként minden más szempontból egyezik tömbökkel a használata, például a `[]` operátorral elérhetők az egyes elemek. Fontos, hogy az egyetlen elemet tartalmazó `tuple` nem ekvivalens a benne tárolt egyetlen elemmel (azaz `("SOA",) != "SOA"`), és mivel a zárójel nyelvi elem, ezért egyetlen elem esetén a plusz vessző jelzi, hogy itt nem csak tagolásról van szó, hanem `tuple`-ről.
A Python nyelven írt kód modulokba szerveződik, egy `.py` kiterjesztésű fájl egy modulnak felel meg, melynek neve megegyezik a fájl kiterjesztés nélküli nevével. A modulok egyrészt importálhatók név szerint, másrészt importálhatók belőlük tetszőleges függvények, osztályok, változók, konstansok. A modul karakterkódolása tetszőleges lehet, viszont ASCII-n kívüli (pl. ékezetes) karakterek használata esetén ezt jelezni kell egy kommentben, (BOM nélküli) UTF-8 esetén például az alábbi módon.

```
# -*- coding: utf-8 -*-
```

2.2. Adatbázis-elérés: DBAPI

A Python adatbázis API-ja a JDBC-hez hasonló módon biztosítja, hogy a különféle adatbázis-kezelőket Python nyelven egységes felületen keresztül lehessen elérni. Fontos eltérés azonban, hogy a kapcsolódás és a paraméteres lekérdezések kezelése meghajtónként különbözhet – egyes adatbázis-kezelők ugyanis például csak pozícionális paraméterezést támogatnak (a JDBC például csak ilyet támogat, adatbázis-meghajtótól függetlenül), míg mások nevesített paraméterek használatát is lehetővé teszik. Az adatbázis-kapcsolatot reprezentáló objektum adatbázis-függő módon jön létre, fontosabb elvárt metódusai és tulajdonságai a következők.

- `cursor()`: visszaad egy adatbázis-kurzort, lekérdezések végrehajtására, azok eredményének elérésére
- `begin()`: tranzakciót indít
- `commit()`: commitolja a tranzakciót
- `rollback()`: rollbackeli a tranzakciót
- `close()`: lezárja a kapcsolatot
- `autocommit`: jelzi, hogy aktív-e az automatikus commit

A `cursor` által visszaadott kurzor objektum a következő metódusokat és tulajdonságokat biztosítja.

- `execute(sql, ...)`: végrehajt egy SQL lekérdezést, opcionálisan több paramétert is fogadhat, melyek meghajtó-függő módon kerülnek értelmezésre paraméteres lekérdezések esetén
- `executemany(sql, params)`: végrehajt egy SQL lekérdezést a második paraméter (ami például egy `list` lehet) összes elemére, ezáltal megspórolva az utasítás ismételt feldolgozását
- `fetchone()`: visszaad egy sort az eredményhalmazból `tuple` típusú n-esként, vagy `None`-t, ha már nincs kiolvasásra váró sor
- `fetchall()`: visszaadja az eredményhalmaz összes sorát `tuple` típusú n-esek `list` tömbjeként
- `close()`: lezárja a kurzort
- `rowcount`: kiolvasható belőle az érintett sorok számossága[23]

Ezen kívül a kiterjesztett DB API-t támogató kurzor objektumok (ilyen az Oracle is) iterálhatók, így például a következő `for` ciklussal kiíratható egy tábla tartalma:

```
cur.execute("SELECT keresztnev, vezeteknev FROM szemely")
for k, v in cur:
    print "Keresztnév:", k
    print "Vezetéknév:", v
```

A fenti példa a Python `for` utasításának két jellegzetes tulajdonságára is rámutat.
Egyrészt a `for` mindig valamilyen iterálható (vö. Java `Iterable<E>` ill. .NET `IEnumerable<E>` interfészei) objektumon (`list`, adatbázis eredményhalmaz) halad végig a ciklussal, hasonlóan a Java új `for` és a C# vagy PHP `foreach` utasításához.
Másrészt amennyiben a ciklus aktuális iterációjában lekért érték több elemből áll (mint a példában az eredményhalmaz egy sora), nem szükséges kézzel elemeire szedni. Amennyiben a `for` és `in` kulcsszavak közt felsorolt változónevek száma megegyezik az aktuális érték elemszámával, 1:1 behelyettesítés történik, ellenkező esetben kivételt kapunk.
A fenti példa tehát leírható lett volna `for row in cur:` kezdettel is, ekkor `k` helyett `row[0]`, `v` helyett `row[1]` kifejezést kellett volna írni.

2.3. Kapcsolódás Oracle-höz: cxOracle

A cxOracle egy Pythonban és C-ben írt, a Python adatbázis API-val kompatibilis modul, melynek segítségével a hivatalos Oracle natív könyvtárak használatával lehet Python kódból Oracle adatbázishoz kapcsolódni. Hasonlóan a JDBC-hez, kapcsolódáskor a felhasználónév, jelszó és SID mellett az adatbázis elérhetőségét kell megadni, mint az az alábbi példában látható.

```
import cx_Oracle

conn = cx_Oracle.connect('felhasznalonev', 'jelszo',
        'szerver.hosztnev.hu:1521/SID')
conn.close()
```

A `connect` által visszaadott, kapcsolatot reprezentáló objektum `close` metódussal zárható le, `cursor` metódussal pedig kurzor kérhető tőle, melyen keresztül a Python adatbázis API-nak megfelelő hívásokkal SQL lekérdezések indíthatók az adatbázis felé. JDBC-től eltérően a Python adatbázis API paraméteres lekérdezéseknél engedi az adatbázisonként eltérő hívási

[23] Meghajtó-függő az implementáció, többek között a következő alfejezetben bemutatott cxOracle meghajtó is csak adatmódosító utasítások (DMS: `DELETE`, `INSERT`, `UPDATE`) esetén támogatja, `SELECT` esetén lehet 0 akkor is, ha nem üres az eredményhalmaz.

konvenciót, ez Oracle esetében mind a lekérdezés szövegében, mind az `execute()` hívásakor nevesített paraméterekkel történik.

```
cur = conn.cursor()
cur.execute('SELECT foo, bar FROM tabla WHERE id = :id', id=42)
results = cur.fetchall()
cur.close()
```

A meghajtó támogatja a Python `with` kontextus-kezelőjét (mely leginkább a C# `using` ill. a Java 7 try-with-resources megoldásához hasonlítható) kurzorok esetében, a `with` blokkon belüli utasítások előtt tranzakciót indít, majd sikeres végrehajtás esetén commitol, kivétel esetén rollbacket hajt végre. Az alábbi két példa tehát ekvivalens.

```
with conn:
    cur.execute('INSERT INTO laborok (labor, sorszam) VALUES ("Szoftlab", 5)')
```

```
try:
    conn.begin()
    cur.execute('INSERT INTO laborok (labor, sorszam) VALUES ("Szoftlab", 5)')
    conn.commit()
except:
    conn.rollback()
    raise
```

Amennyiben a rekord azonosítóját trigger tölti ki (például szekvencia alapján), a következő módon tudjuk megszerezni a beszúrt sor azonosítóját (forrás, természetesen ez a megoldás nemcsak a rekord azonosítójára, hanem bármely mezőjének a ténylegesen beszúrt értékének a lekérdezésére használható):

```
idVar = cursor.var(cx_Oracle.NUMBER)
cursor.execute("""INSERT INTO tabla (oszlop) VALUES (:ertek)
    RETURNING azon INTO :az""", ertek=42, az=idVar)
azon = idVar.getvalue()
```

2.4. Dátum és idő kezelése

A Python standard könyvtárában szereplő `datetime` modul `datetime`, `date` és `time` osztályai használhatók dátumok, időpontok és ezek kombinációinak reprezentálására. Ennek megfelelően a cx_Oracle is ilyen objektumokkal tér vissza megfelelő típusú oszlop esetén, és paraméterként is elfogad ilyet. A JSON formátum azonban nem tartalmaz ilyet, így itt ad-hoc módszerek terjedtek el, leggyakoribb az ISO 8601 szabványú string használata – ezt ismertetjük alább is – valamint találkozhatunk még Unix időbélyeg (az 1970. január 1. óta eltelt másodpercek számának) számként való reprezentálásával.

Dátum (és dátum-idő) ISO 8601-be történő konvertálására beépített támogatással rendelkezik a modul; mind `datetime`, mind `date` objektumok esetén az `isoformat()` metódus közvetlenül a megfelelő formátumot adja vissza, opcionális paraméterként megadható a dátumot és időt elválasztó paraméter (alapértelmezésben „T” betű, de elterjedt még a szóköz is ilyen célra). Ellenkező irányban (string parse-olása, pl. JSON-ből adatbázisba íráshoz) összetettebb a feladat, a `datetime` objektum statikus `strptime(string, format)` metódusa egy megadott formátumsztring szerint olvassa be a bemenetet és állít elő `datetime` objektumot.

Természetesen `datetime` példányból leválasztható akár a dátum, akár az idő a `date` ill. `time` metódusok visszatérési értékeként, és mindhárom típusra működnek a standard összehasonlító operátorok (=, <, >, stb.).

2.5. Webszolgáltatás keretrendszer: Flask

A Flask egy Python nyelven írt webes mikro-keretrendszer, mellyel webalkalmazások és webszolgáltatások is készíthetők. Utóbbi szempontjából segítség, hogy készen tartalmaz a HTTP kérések mintaillesztés-alapú kiszolgálására diszpécser logikát, a fejlesztést pedig gyorsítja, hogy külső web- vagy alkalmazásszerver (Apache, Tomcat, IIS) nélkül is működőképes és tesztelhető az elkészült szolgáltatás. Használata az alábbi példa alapján elsőre átlátható:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def szoftlab5():
    return "Szoftlab 5"

if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

Az első két sorban importáljuk a `flask` modulból a `Flask` osztályt, majd példányosítjuk `app` néven. Ezt követően ezen a példányon keresztül építhetjük fel a webalkalmazást, a `route` dekorátort használva. Ennek egyetlen kötelező paramétere az az útvonal, melynek HTTP-n való lekérése esetén a megjelölt függvény fog lefutni. A függvény visszatérési értékéből a Flask előállítja a HTTP választ, string visszaadott típus esetén `text/html` megjelöléssel.

Végül az utolsó két sor fejlesztés-tesztelés során hasznos, amennyiben a modult közvetlenül indítjuk (pl. parancssorból `python modulneve.py`), elindítja a Flask beépített webszerverét az 5000-es TCP porton, a hibakeresőt engedélyezve (utóbbit persze éles üzemben nem ajánlott bekapcsolni). Ennek köszönhetően a fenti szkriptet elindítva a böngészőben megnyitható a http://localhost:5000/ URL, melyen a `"Szoftlab 5"` szöveg köszönt minket.
A `__name__` ugyanis az aktuális modul nevét tartalmazza (azaz általában a Python forrásfájl neve kiterjesztés nélkül), kivéve, ha az adott fájl a program belépési pontja, ekkor ugyanis a `"__main__"` értéket kapja. A `Flask` osztály konstruktorának belső működéséhez van szüksége a webalkalmazás/webszolgáltatás fő moduljának nevére, ezért kerül ez átadásra, az utolsó blokkban pedig így oldható meg, hogy a fájl közvetlenül indítva elinduljon a fejlesztői szerver, más modulból (például web- vagy alkalmazásszerverből) importálva viszont ne.
REST webszolgáltatások készítésekor hasznos segítség a `flask` modul `jsonify` függvénye, melynek visszatérési értéke egy olyan HTTP válasz, amely JSON formátumban tartalmazza a függvény paramétereit, és megfelelően be is állítja a `Content-Type` fejlécet. Használata az alábbi példából látható (a beépített webszervert indító részt ezúttal elhagytuk):

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/lab.json")
def lab():
    return jsonify(szoftlab=5, laborok=["Oracle", "SQL", "JDBC", "XSQL", "SOA"])
```

A fenti példát futtatva a következő válasz érkezik a http://localhost:5000/lab.json URL-re irányuló kérésre:

```
HTTP/1.0 200 OK
Content-Length: 98
Content-Type: application/json
Date: Sun, 05 Apr 2015 09:06:03 GMT
Server: Werkzeug/0.10.4 Python/2.7.9
```

```
{
    "laborok": [
            "Oracle",
            "SQL",
            "JDBC",
            "XSQL",
            "SOA",
    ],
    "szoftlab": 5
}
```

A diszpécser logika lehetővé teszi paraméterezett útvonalak létrehozását is, ilyenkor az útvonalban kisebb-nagyobb jelek közé írt azonosítóval jelölhetjük meg a változó bemenet helyét és nevét. A `route` dekorátorral megjelölt függvény fejlécében ezután minden azonosítóhoz kell tartozzon egy azonos nevű paraméter is, ebben kerül ugyanis átadásra a mintaillesztés eredménye, ezt a következő példa szemléletesen bemutatja (itt az import részt is elhagytuk).

```
@app.route("/laborok/<n>.json")
def labor(n):
    return "Szoftlab {0}".format(n)
```

A fenti példát futtatva a következő válasz érkezik egy, a http://localhost:5000/laborok/5.json URL-re irányuló kérésre:

```
HTTP/1.0 200 OK
Content-Length: 10
Content-Type: text/html; charset=utf-8
Date: Sun, 05 Apr 2015 09:07:47 GMT
Server: Werkzeug/0.10.4 Python/2.7.9

Szoftlab 5
```

A beépített diszpécser miatt (pl. PHP, ASP, CGI megoldásokkal szemben) nem fájlrendszer alapján történik a kérések kiszolgálása, így statikus fájlok kiszolgálásához az alapértelmezett Flask konfigurációval az aktuális könyvtár `static` alkönyvtárát lehet használni. Az ezen belül található fájlok a `/static/` útvonalon belül kerülnek kiszolgálásra, tehát például a `static` könyvtáron belül elhelyezett `foo.png` kép a beépített webszervert használva a `http://localhost:5000/static/foo.png` URL-en érhető el. Bár tetszőleges plusz könyvtár felvehető, ezt a mintát követve később, éles környezetben egyszerűen megoldható, hogy a statikus fájlok különálló, erre optimalizált szerverről (lighttpd, nginx) kerüljenek kiszolgálásra.

2.6. Távoli webszolgáltatások elérése: Requests

A Requests egy Python nyelven írt, fejlett HTTP kliens könyvtár, melynek funkcionalitása REST szolgáltatások igénybevételét is megkönnyíti. A modul HTTP verbekkel egyező nevű függvényeivel egyszerűen elérhető bármilyen webes forrás, ezek visszatérési értéke pedig egy olyan `Response` típusú objektum, melynek metódusain és tulajdonságain keresztül magas szinten kezelhető a HTTP válasz tartalma. Az alábbi, első példában egy egyszerű GET kérést küldünk, majd kiíratjuk a választ a standard kimenetre.

```
import requests

response = requests.get("http://currencies.apps.grandtrunk.net/getlatest/eur/huf")
print response.text
```

A szkript kimenete:

```
298.880251501
```

URL paramétereket a `params` nevű paraméterben átadva a könyvtár magától elvégzi a megfelelő kódolási feladatokat, megfelelő `Content-Type` érték esetén pedig a JSON formátumú válaszok feldolgozásra kerülnek, és strukturált módon elérhetők a válasz `json()` metódusán keresztül. A következő példa az *OpenStreetMaps Nominatim* szolgáltatásán keresztül lekérdezi, majd kiíratja Dobogókő koordinátáit. A válasz JSON szótárak listája, melynek minden elemén végigiterálunk, majd kiíratjuk a `lat` és `lon` elemét. Látható, hogy a *Requests* a JSON szótárból *Python* `dict`-et épített, mely a tömbhöz hasonlóan címezhető, hasonlóan, mint a PHP asszociatív tömbjei, a *Java* `Map` vagy a *C#* `IDictionary` interfésze esetében.

```
params = {'format': 'json', 'city': u'Dobogókő', 'country': 'Hungary'}
response = requests.get('http://nominatim.openstreetmap.org/search',
params=params)
results = response.json()
for result in results:
    print result['lat'], result['lon']
```

A szkript kimenete:

```
47.7193734 18.8959654
```

3. Böngészőoldal: JavaScript

3.1. Programnyelv: JavaScript

A Pythonhoz hasonlóan a *JavaScript* is egy olyan programozási nyelv, mely lehetővé teszi – akár egy programon belül is – több (funkcionális, procedurális, objektum-orientált) paradigma használatát is. Az elterjedt böngészők mindegyike támogatja weboldalakba illesztését, azonban ma már szerveroldalon is használják. A böngészők és azok egyes verziói között kisebb-nagyobb implementációbeli különbségek tapasztalhatók a webes funkcionalitás tekintetében, így gyakran JavaScript keretrendszerek (*jQuery*, *Prototype*, stb.) egészítik ki a beépített könyvtárak lehetőségeit. Szintaktikájában a *Java* nyelvre hasonlít, annak típus-meghatározásai nélkül.

A nyelv dinamikusan típusos, a változók típusát nem szükséges előre deklarálni, az értékadáskor kerül meghatározásra. A következő kódrészlet végére például `b` lehet `Number` vagy `String` típusú is.

```
if (a > 5) {
    b = 10;
} else {
    b = "kisebb";
}

alert(b);
```

A beépített típusok a következők; a példákban látható, hogyan adható meg kódban egy adott típusú konstans érték:

- `Null`: null érték, pl. `null`
- `Undefined`:inicializálatlan változók és függvény-paraméterek, pl. `undefined`
- `Number`: számok, pl. `42`, `5.5`
- `Boolean`: logikai érték, pl. `true`, `false`
- `String`: karakterekből álló Unicode string, pl. `u"M\u0171egyetemi hallgató"`
- `Array`: tömb, pl. `[42, 5.5, "szoftlab5"]`
- `Object`: szótár és objektum, pl. `{"telefon": "phone"}`, `{1: "Jan", 2: "Feb"}`

Látható, hogy hogy – az undefined kivételével – a típusok reprezentációja megegyezik a JSON-nel, amely pont ezért lett népszerű olyan környezetekben, ahol JavaScript kód is érintett a feldolgozásban.

3.2. Keretrendszer és lehetőségek: jQuery és AJAX

A weboldalakba ágyazott JavaScript egyik felhasználói élményt javító lehetősége az oldal adatainak, megjelenítésének frissítése a teljes oldal újra betöltése nélkül. Ennek fejlődése során fontos lépés volt, hogy a JavaScript a háttérben HTTP kéréseket indíthasson az oldal újratöltése nélkül, ezt 2005-ben a *Google Suggest* tette igazán népszerűvé. Mivel eleinte XML volt a felhasznált formátum, a megoldás AJAX (*Asynchronous JavaScript and XML*) néven terjedt el, azonban semmi sem korlátozza a felhasználható formátumok körét, így a JSON is felzárkózik az XML mögé e területen.
Bár AJAX megoldás készíthető közvetlenül a böngésző API-jának felhasználásával, egyszerűbb és stabilabb megoldást jelent egy JavaScript könyvtár használata. A laboron a jQuery használatát mutatjuk be. A jQuery használatához a HTML dokumentum fejlécében (a `<head>` részben) a következő sort kell beszúrni – az `integrity` attribútum kriptográfiai hashfüggvény (256-bites SHA-2) segítségével biztosítja, hogy a külső félnél (*jquery.com*) elhelyezett aktív, így weboldalunk működését befolyásoló tartalmat ne cserélhesse le rosszindulatú személy észrevétlenül:

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js" integrity="sha256-
ZosEbRLbNQzLpnKIkEdrPv7lOy9C27hHQ+Xp8a4MxAQ=" crossorigin="anonymous"></script>
```

AJAX megoldás készítéséhez három jQuery funkciót érdemes ismerni. Az első az oldal betöltődésekor lefutó kód beállítását teszi lehetővé, a `$()` nevű függvény segítségével. Az alábbi kód az oldal betöltődését követően feldob egy „Szoftlab5” üzenetablakot.

```
<script>
    $(function() {
        alert("Szoftlab5");
    });
</script>
```

A megoldás jól szemlélteti a JavaScript funkcionális jellegét, ugyanis egy függvényt (`function() { ... }`) adtunk át paraméterként egy másik függvénynek (`$()`). A kódot természetesen írhattuk volna közvetlenül a `<script>` tagek közé is, ekkor azonban a kód már akkor lefutott volna, amikor a böngésző az adott `<script>` taghez ér, így például nem hivatkozhat a dokumentum további részében definiált objektumokra, valamint az oldal betöltését is lassítja (a példánál maradva, amíg a felhasználó nem zárja be az üzenetablakot, az oldal betöltése szünetel). A `$()` függvény tehát amennyiben egyetlen paramétereként egy másik függvényt kap, annak végrehajtását akkorra időzíti, mikor a teljes weboldal betöltődött a böngészőbe.
A második hasznos funkcionalitás az ún. szelektor (amivel CSS stíluslapok írásakor már néhányan találkozhattak), amellyel HTML elemek egyszerűen kiválaszthatók későbbi manipuláció céljára. Erre is a `$()` függvény használható, ám ezúttal `String` paraméter kerül neki átadásra. A visszatérési értéken tetszőleges, az elemet érintő művelet elvégezhető, a `html()` metódus például a paramétereként kapott stringre cseréli az elem tartalmát.
Az alábbi példában felhasználjuk a fent megismert technikát is arra, hogy hivatkozhassunk a `mezo` ID-jű elemre úgy, hogy a `<script>` blokk előbb kerül definiálásra. Amennyiben tehát a `$()` függvény `String` típusú paramétert kap, kiértékeli a benne lévő CSS-szerű kifejezést, majd visszaad egy, a kifejezésre illeszkedő elemeket reprezentáló objektumot. A felesleges ciklusok elkerülésére az ezen végrehajtott metódusok minden illeszkedő elemet érintenek, így

például ha a lenti kódban a szelektor több elemre is illeszkedne, mindegyik tartalmát a megadott szövegre módosítaná.

```
<script>
    $(function() {
        $("#mezo").html("jQuery a HSZK-ban");
    });
</script>
<div id="mezo">(egyelőre üres)</div>
```

A harmadik szükséges építőelem maga az AJAX hívás, melyet a `$` nevű objektum `ajax` metódusával érhetünk el. Ennek visszatérési értéke nem a kapott érték, hiszen az aszinkron működés lényege épp az, hogy a hívás azonnal visszatér, viszont megadhatunk eseménykezelőket, amelyek meghívásra kerülnek például sikeres AJAX kérés esetén. Az `ajax` metódus egy objektumot vár, ennek gyakran használt paraméterei az alábbi példában kerülnek bemutatásra. Fontos, hogy biztonsági okokból ezzel a megoldással külön engedély nélkül csak a saját weboldalunkon belülre indíthatunk lekéréseket, ezt az ún. *Same Origin Policy* szabályozza.
Az alábbi példában három függvényhívás történik, ezeket egymásba ágyazva definiáltuk, kihasználva a JavaScript lehetőségeit. A külső `$(...)` hívás az első jQuery példában látott módon beállít egy függvényt, hogy akkor fusson le, amikor az oldal betöltődött. Ennek törzse a `$.ajax(...)` hívás, mely aszinkron kérést indít a `/service.json` címre, a választ pedig `json` típusként kérjük, hogy értelmezze. Mivel a függvény azonnal visszatér, definiálunk egy eseménykezelő függvényt, mely egyetlen paramétert vár, ebbe kerül a válaszként kapott adat. Ennek megfelelően a `data` paraméter változót kizárólag a legbelső függvényben törzsében érhetjük el, ez akkor fog lefutni, amikor az AJAX kérésre válasz érkezett.

```
<script>
    $(function() {
        $.ajax({
            url: "/service.json",
            dataType: "json",
            success: function(data) {
                $("#mezo").html(data.lab);
            }
        });
    });
</script>
<div id="mezo">(egyelőre üres)</div>
```

Ha például a `/service.json` válasza `{"lab": "szoftlab5"}`, a `mezo` ID-jű elembe a *szoftlab5* szöveg kerül. Fontos, hogy amennyiben a JSON szintaktikai hibát tartalmaz, a kérés hibajelzés nélkül meghiúsul, így előbb mindig bizonyosodjunk meg róla, hogy a JSON szintaktikailag helyes-e. Az egyszerűbb érthetőség kedvéért az alábbi példa viselkedésében ekvivalens, viszont a függvényeket kifejtve tartalmazza.

```
<script>
    function success_handler(data) {
        $("#mezo").html(data.lab);
    }

    function page_loaded() {
        $.ajax({
            url: "/service.json",
            dataType: "json",
            success: success_handler
        });
    }
```

```
    $(page_loaded);
</script>
<div id="mezo">(egyelőre üres)</div>
```

3.3. JavaScript és AJAX hibakeresés

Amennyiben a böngészőben futó szkriptjeink nem a várt eredményt adják, több lehetőség is van a belső működés megismerésére. A legegyszerűbb módszer az `alert()` beépített függvény használata, mely az egyetlen paraméterét felugró ablakban megjeleníti stringként. Ennek megfelelően hasznossága is korlátozott, főleg, ha sok és/vagy strukturált adat megjelenítése szükséges. Jobb módszer a böngészőbe épülő debuggerek használata, ebből a Google Chrome / Chromium debuggerét mutatjuk be, mivel ez régóta a böngészőbe építve érkezik és sok operációs rendszerre elérhető. Hasonló funkcionalitást elérhető természetesen Mozilla Firefox és Microsoft Internet Explorer böngészők újabb verzióiban is és/vagy bővítmények használatával.

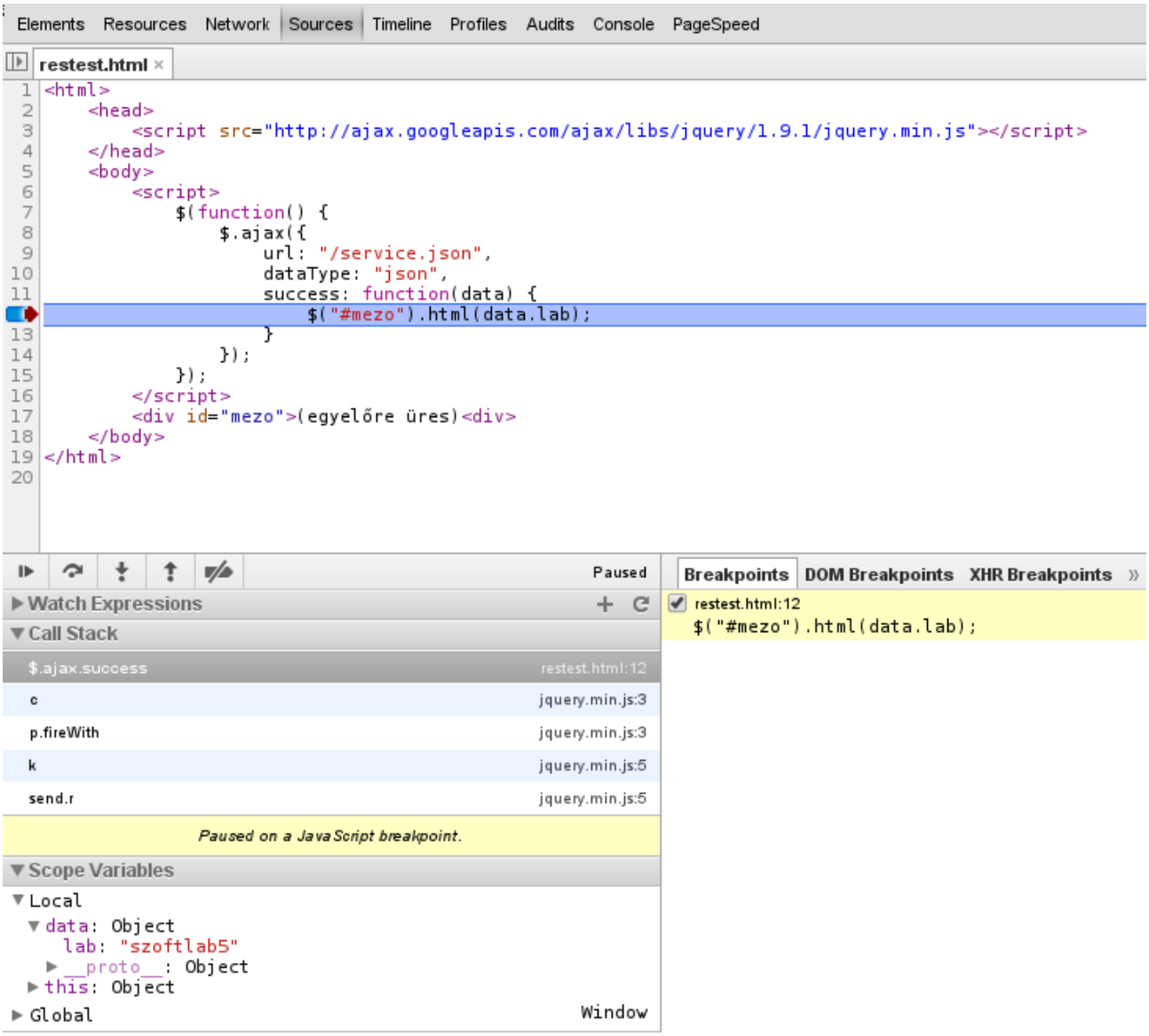
A Chrome debuggere az F12 gomb megnyomásával indítható, választható, hogy a böngészőablak aljához vagy jobb oldalához legyen dokkolva, utóbbi szélesvásznú képernyőkön lehet kényelmesebb. A debugger felső részén találhatók a funkcionalitást rejtő fülek, ebből számunkra négy lesz fontos:

- Elements: az éppen aktuálisan megjelenített HTML oldal szerkezete nézegethető és változtatható benne (elemek törölhetők, módosíthatók, hozzáadhatók) eltérően a forrás megjelenítése ablaktól, mely a betöltéskori állapotot mutatja, amelyet a szervertől kapott
- Network: a hálózatról érkezett válaszokat mutatja, megtekinthető a betöltéshez szükséges idő, illetve a kérések és válaszok tartama és fejlécei is
- Sources: a betöltött szkriptek forrása tekinthető meg itt, töréspontok állíthatók be, szokásos debugger funkcionalitás JavaScripthez
- Console: külön is felnyitható az alsó sávon található *Show console* gombbal, a Python interpreterhez hasonlóan interaktív lehetőséget ad JavaScript kifejezések kiértékelésére

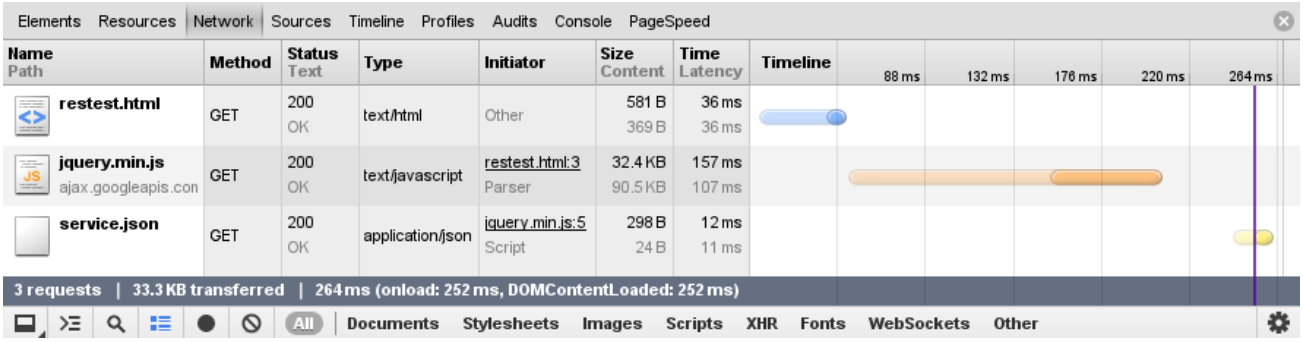
JavaScript hibakeresésnél már a debugger kinyitásakor értesülhetünk arról, ha hibát észlelt a Chrome a futtatáskor, ezt a jobb alsó sarokban egy piros alapon fehér × jelzi, a mellette lévő szám a hibák számát jelenti. Erre kattintva megtekinthető a hibák pontos helye és a hibaüzenet szövege. Amennyiben ebből sem világos a hiba oka, érdemes az összetettebb kifejezéseket akár a Console fülön kipróbálni, akár töréspontot beállítani az adott sorra.

Töréspont (*breakpoint*) a forráskódot megnyitva a bal szélső sorszámra kattintva állítható be és törölhető, ezt követően amint a futás az adott sorra ér, a weboldalon a *Paused in debugger* üzenet látható, az adott sor kiemelésre kerül, és a forrás alatt láthatók a helyi és globális változók értékei és a hívási verem (*call stack*).

A futás ekkor a forráskód alatti gombokkal folytatható (play-szerű gomb), vagy akár lépésenként folytatható, így soronként figyelhető, hogyan viselkedik a kód. Az alábbi ábrán látható például a debugger állapota az AJAX példa `success` eseménykezelőjére állított töréspont esetén. Alul megfigyelhető a `data` paraméter értéke, a JSON-ből dekódolt objektum egyes attribútumai a nyilakra kattintva egyenként kibonthatók.



AJAX hibakeresésnél hasznos még a Network fül, melyben az összes elküldött kérés látható néhány alapadatával együtt. Jobb oldalon megfigyelhető, melyik kérés mennyi időt vett igénybe, így hamar kideríthetők a függőségek (amíg X nem töltődött be, addig Y lekérése el sem indult) és hatékonyabban gyorsítható az oldal betöltődése a lassító elemek felismerésével. Piros és kék függőleges vonal jelzi a betöltött oldal elkészültségét jelző két esemény, alább látható a fenti AJAX példa betöltése után a Network fül állapota. Látszik, hogy a `/service.json` kérés csak az oldal betöltését követően került elküldésre, a 32 kB méretű jQuery pedig összesen 157 ms alatt töltődött be, ebből 107 ms (hálózati) késleltetés.



Az ablak alján lehet szűrni az egyes betöltött elemekre típusuk szerint, alapértelmezés szerint minden megjelenik (*All*), de például XHR-t kiválasztva lehet kizárólag AJAX-szal lekért elemekre szűrni (*XmlHttpRequest*). AJAX hibakeresésnél az ablak több szempontból lehet hatékony segítség: egyrészt megismerhető, hogy egyáltalán lekérésre került-e a kért erőforrás, a lekérés sikeres volt-e, és ha igen, milyen tartalom érkezett meg. Utóbbit az erőforrás nevére (bal oszlop) kattintva lehet megnézni, ekkor a többi oszlop helyén négyfüles ablak jelenik meg.

- Headers: a HTTP kérés és válasz fejlécei tekinthetők meg
- Preview: a válasz feldolgozott formában tekinthető meg, JSON, XML, HTML esetében például faszerkezetben
- Response: a válasz feldolgozatlan, nyers állapotban tekinthető meg
- Timing: az adott erőforrásra vonatkozó várakozási idő

V. labor: XML alapú adatkezelés

Írta: Mátéfi Gergely
Nagypál Gábor, Bihari István, Hajnács Zoltán korábbi segédletének felhasználásával

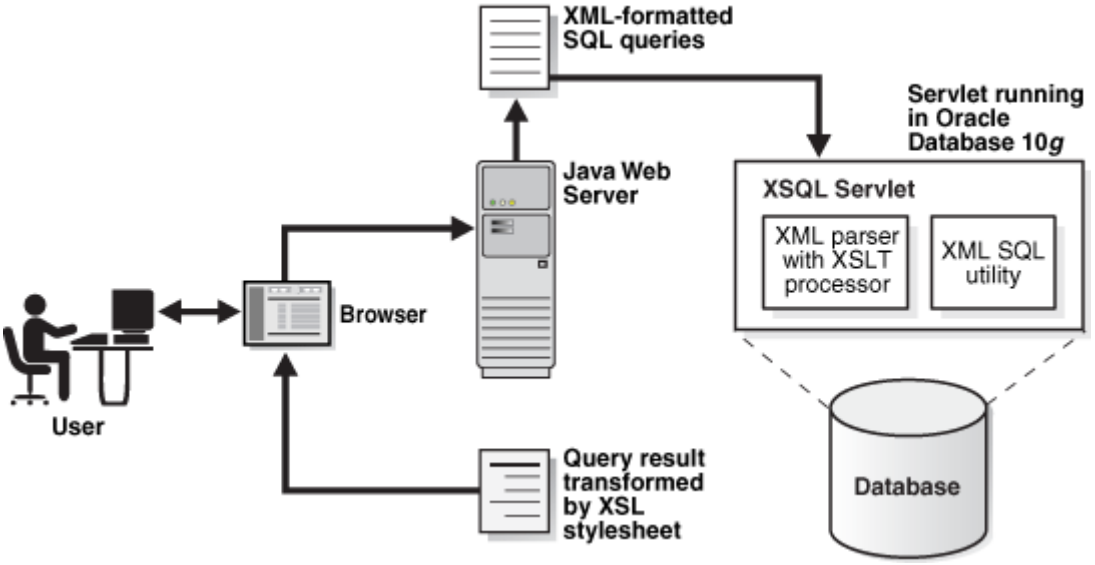
1. Bevezetés

A nagyobb informatikai rendszerek jellemzően több hozzáférési felülettel rendelkeznek, például vastagklienssel, webes megjelenési felülettel, adatkapcsolati interfésszel külső informatikai rendszer felé, stb. Több hozzáférési felület mellett a hagyományos kliens-szerver architektúra nem hatékony, ehelyett az összetettebb rendszerek felépítése *többrétegű architektúrát* követ. Többrétegű architektúra esetén az adatelemekre vonatkozó előírások betartatásáért felelős *alkalmazáslogikai réteg* és a felhasználói felület kezeléséért felelős *megjelenítési réteg* szétválik, és a különböző hozzáférési felületek a közös alkalmazáslogikai réteget használják.

Az egyes rétegekben található, esetenként eltérő platformon futó, eltérő gyártótól származó szoftverkomponensek integrációjához az adatelemek leírására alkalmas közös nyelv szükséges. Az elmúlt években az Extensible Markup Language (XML) nyelv vált az adatelemek leírásának de facto szabványává. Az XML platformfüggetlen, szöveges formátumú jelölő nyelv, amely alkalmas információk és adatelemek struktúrált leírására és továbbítására. Kapcsolódó szabványai lehetőséget teremtenek XML sémák definiálására és validálására, valamint a különböző sémák közötti transzformációra is.

Jelen labor célja, hogy bepillantást engedjen az XML alapú adatbázis-alkalmazásfejlesztésbe. Az első alfejezet XML dokumentumok szerkezetét mutatja be, ezt követően megnézzük, hogyan transzformálható egy XML dokumentum az XSLT és XPath technológiák segítségével. A segédlet az XML 1.0, XSLT 1.0 és XPath 1.0 szabványokra épül. A

technológiai elemek adott környezetben való alkalmazásának folyamatát az alábbi ábrán lehet követni.



2. XML dokumentumok felépítése

2.1. Elemek és Címkék

Az XML dokumentumok, a HTML-hez hasonlóan, szöveges formátumú fájlok, amelyek egymásba ágyazott elemeket tartalmaznak. Az elemeket nyitó- és zárócímkék (*tagek*) jelölik, mint ahogy az alábbi példa is mutatja:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<uzenet>
  <felado>Géza</felado>
  <cimzett>Béla</cimzett>
  <torzs>Szia Világ!</torzs>
</uzenet>
```

A HTML-lel szemben az XML szabvány nem rögzíti a címkeszótárat. Ehelyett az alkalmazástól függő mindenkori nyelvtan (XML séma) határozza meg, milyen címkék, milyen egymásba ágyazási szabályokkal szerepelhetnek a dokumentumban. Az XML terminológiája szerint egy dokumentum *jól formált* (well-formed), ha szintaktikája betartja az XML előírásait, és *érvényes* (valid) egy sémára nézve, ha követi annak nyelvtanát.
A címkék neve betűvel vagy aláhúzással („_”) kezdődhet, nem lehet benne szóköz és nem kezdődhet „xml”-el. Az XML a címkék nevében különbséget tesz kis- és nagybetűk között, tehát például a `<Felado>` és a `<felado>` címkék különbözőnek számítanak.

2.2. Szerkezet

Minden XML dokumentum egy vezérlési utasítással kezdődik, amely kötelezően tartalmazza az XML verziószámát, és opcionálisan a felhasznált kódlapot:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
```

Az XML dokumentumok elemei fa struktúrát alkotnak. Egy dokumentumnak pontosan egy gyökér eleme kell, hogy legyen; ez a fenti példában az `<üzenet>` elem volt. Minden nyitócímkéhez kell, hogy tartozzon egy záró címke is, amelyek közrefogják az elemhez tartozó adatot. Ellentétben a HTML (csak a gyakorlatban!) megengedőbb szabályaival, XML-ben a nyitó- és zárócímkéknek követniük kell a zárójelezési szabályokat. Így például az alábbi forma működhet HTML-ben, de bizonyosan szabálytalan XML-ben:

```
<b>Ez vastag <i> és ez még dőlt is </b> ez már csak dőlt </i>
```

Helyette a szabályos XML leírás:

```
<b>Ez vastag <i> és ez még dőlt is </i></b><i> ez már csak dőlt </i>
```

Ha egy elemhez nem tartozik adat, akkor a nyitó- és a zárócímkék összevonhatóak egy önlezáró címkébe (empty-element tag), például a `<br></br>` helyett egyszerűen írható `<br/>`. A megjegyzéseket, a HTML-hez hasonlóan, a `<!-- Megjegyzés -->` szintaktikával jelezhetjük egy XML dokumentumban. A megjegyzések nem ágyazhatók egymásba.

2.3. Attribútumok

Az XML logikája szerint egy elem tulajdonsága vagy gyermekelemmel, vagy attribútummal írható le, a használt séma szabályainak megfelelően. Az attribútumokat a nyitócímkébe lehet elhelyezni, például:

```
<üzenet kelt='20050221'>Hello!</üzenet>
```

Az attribútum értéke megadható mind aposztróf, mind idézőjel határolók között, azonban – ellentétben a HTML-lel – a határolót nem szabad elhagyni. Egy attribútum egy címkében csak egyszer szerepelhet.

2.4. Helyettesítő szekvenciák

A címkékkel jelölt adatokban bármilyen karakter szerepelhet, kivéve a foglalt `<` és `&` vezérlőkaraktereket. A vezérlőkarakterek helyett az XML helyettesítő szekvenciák használhatóak, lásd az alábbi táblázatban. Például ez szabálytalan megadás:

```
<formula>a < b & b < c => a < c </formula>
```

Helyette ez használandó:

```
<formula>a &lt; b &amp; b &lt; c => a &lt; c</formula>
```

Alternatívájaként a speciális `<![CDATA[...]]>` határoló is alkalmazható:

```
<formula><![CDATA[a < b & b < c => a < c]]></formula>
```

Eredeti karakter	&	<	>	”	’	újsor
Helyettesítés	`&`	`<`	`>`	`"`	`'`	`
`

2.5. Névterek

Mivel a címkeszótárat az alkalmazások határozzák meg, ezért különböző alkalmazásoktól származó dokumentumok összefésülésekor névütközések jöhetnek létre a címkék nevében. A névütközések elkerülésére *névterek* használhatóak. Egy névteret az alkalmazás által definiált *névtér URI* (Uniform Resource Identifier, egységes erőforrás azonosító) azonosít, például a később ismertetendő XSLT a `http://www.w3.org/1999/XSL/Transform` névteret használja. A névtérbe tartozó címkék használatához egy egyedi prefixet kell definiálni a névtér számára az `xmlns:prefix="névtér URI"` speciális attribútummal. Ezt követően a névtér címkéire *kvalifikált elnevezéssel*, a `<prefix:címkenév>` formában lehet hivatkozni, ahogy az alábbi példa is szemlélteti:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<mail:message from="Béla" to="Réka" xmlns:mail="internet:mail">
  <mail:subject>Találka</mail:subject>
  <mail:body xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:body>
      Találkozzunk <xhtml:b>6-kor</xhtml:b> a szokott helyen!
    </xhtml:body>
  </mail:body>
```

```
</mail:message>
```

3. XML dokumentumok létrehozása

Az XML dokumentumok előállításának lehetséges technológiáival a jelen keretek között részletesen nem foglalkozunk, csupán megemlítjük, hogy természetesen lehetősége van pl. SQL utasítások segítségével is kialakítani a tartalmát. Jelen körülmények között feltételezzük, hogy az input XML dokumentum már rendelkezésünkre áll.

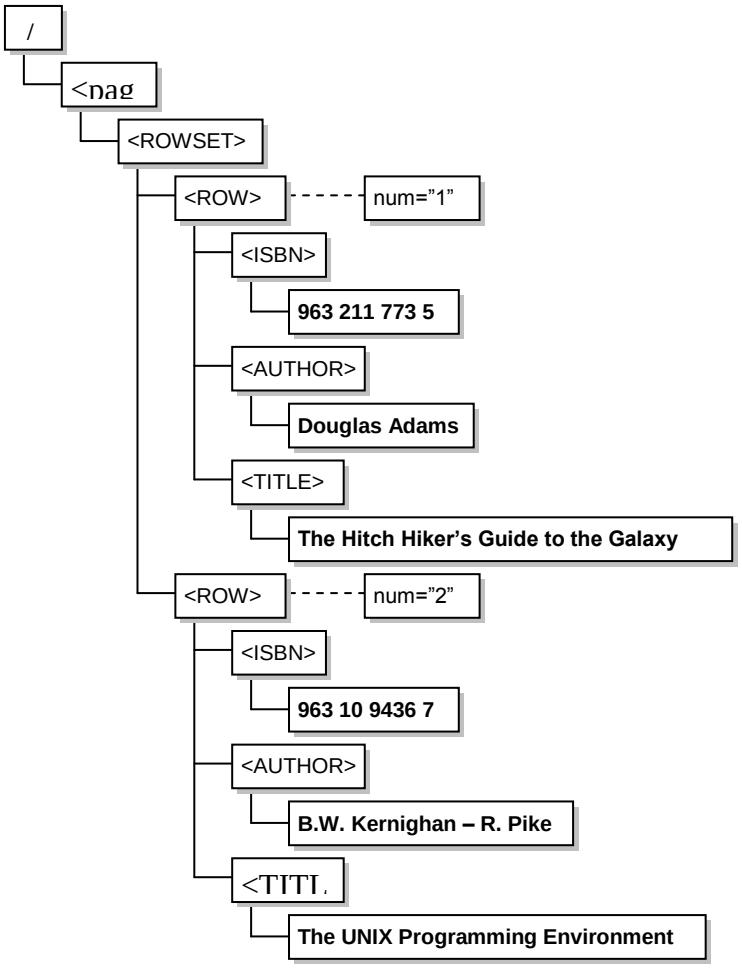
4. Az XSL transzformáció

Az *XSL Transzformáció* (XSLT) bemenő XML dokumentumok fa struktúrájú reprezentációját transzformálja át kimeneti fa struktúrába. Az XSL transzformáció szabályait úgynevezett *XSL stíluslapok* (XSL stylesheets) határozzák meg.[24] *XSLT processzornak* nevezzük azt a szoftvert, amely végrehajtja az XSL transzformációt.
Vegyük a következő, egyszerű könyvtári példát, melyben két sorban egy-egy könyvet kívánunk jellemezni:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<page>
<ROWSET>
  <ROW num="1">
    <ISBN>963 211 773 5</ISBN>
    <AUTHOR>Douglas Adams</AUTHOR>
    <TITLE>The Hitch Hiker's Guide to the Galaxy</TITLE>
  </ROW>
  <ROW num="2">
    <ISBN>963 10 9436 7</ISBN>
    <AUTHOR>B.W. Kernighan-R. Pike</AUTHOR>
    <TITLE>The UNIX Programming Environment</TITLE>
  </ROW>
</ROWSET>
</page>
```

Ennek kanonikus fa-struktúrájú reprezentációját az alábbi ábra szemlélteti. Ezen az XML dokumentum csomópontokból álló faként jelenik meg.

[24] Vigyázat: az XSL stíluslapok nem azonosak a HTML szabványból ismeretes CSS (Cascading Style Sheets) stíluslapokkal!



A következő csomópont típusokat különböztetjük meg: *gyökér csomópont* (root node), *elem csomópont* (element node), *szöveges csomópont* (text node), *attribútum csomópont* (attribute node).[25] A dokumentum minden esetben gyökér csomóponttal kezdődik, amely magát a dokumentumot reprezentálja. A gyökér csomópontnak egyetlen elem csomópont gyermeke lehet, a példán ez a `<page>` csomópont. Elem csomópontoknak további attribútum, szöveges és elem csomópont gyermekei is lehetnek, tetszőleges kombinációban.

Megjegyzendő, hogy az ábra nem teljesen egyezik a korábbi szöveges reprezentációval, amelyet az átláthatóság kedvéért sortörésekkel és tabulátorokkal tagoltunk. A tagolás teljesen szabályos az XML szintaktikában, azonban a tagoló karakterek a fa struktúrában is megjelennek szöveges csomópontokként.

Az XSL stíluslap maga is egy XML dokumentum, amely az XSLT névtérbe tartozó utasításokat használ a transzformáció leírásához. A stíluslapok felépítését az alábbi példa demonstrálja, amelyben a könyvek kanonikus formátumban átadott listáját transzformáljuk HTML táblázatos alakra:

[25] A teljesség kedvéért megemlítjük, hogy létezik még *névtér csomópont* (namespace node), *feldolgozási utasítás csomópont* (processing instruction node) és *megjegyzés csomópont* is (comment node).

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <body>
    <table>
      <xsl:for-each select="page/ROWSET/ROW">
        <tr>
          <td><xsl:value-of select="ISBN"/></td>
          <td><xsl:value-of select="AUTHOR"/></td>
          <td><xsl:value-of select="TITLE"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Az XSL stíluslap gyökéreleme az `<xsl:stylesheet>`. Egy stíluslap egy vagy több *sablont* (template) tartalmazhat. Sablon definiálható a forrás XML dokumentum bármely csomópontjához, ekkor az adott csomópontra illeszkedő sablon határozza meg az adott részfa transzformációját. A sablonokról később még bővebben lesz szó, itt az egyszerűség kedvéért csak egyetlen sablont definiáltunk az `<xsl:template>` paranccsal, amely közvetlenül a gyökérelemre illeszkedik. A példa sablon vegyesen tartalmaz HTML címkéket (`<body>`, `<table>`, `<tr>`, `<td>`), valamint XSLT parancsokat, mint `<xsl:for-each>` és `<xsl:value-of>`. A XSL feldolgozás során az XSLT névtéren kívüli címkék (itt a HTML címkék) változatlan formában kerülnek a kimenetre, ugyanakkor az XSLT parancsok értelmeződnek és végrehajtódnak.

A mintában szereplő `xsl:for-each` XSLT utasítás iterációra szolgál: a végigmegy a `select` attribútumában kiválasztott csomópontokon, és a beágyazott utasításokat minden egyes részfára végrehajtja. A példában a „page/ROWSET/ROW” kifejezés az XML forrásban szereplő ROW elemeket választja ki.

Az `xsl:value-of` XSLT utasítás egy szöveges csomópontot hoz létre a kimeneten. A szöveges csomópont tartalmát a `select` attribútumában megadott kifejezés határozza meg. A példában szereplő „ISBN”, „AUTHOR”, „TITLE” kifejezések a feldolgozás során éppen aktuális részfák értékét, azaz a keresett mezők értékét adják vissza.

A transzformáció befejeztével az eredményfát az XSLT processzor szöveges formában írja a kimenetre. Ezt a folyamatot nevezzük az eredményfa *szerializációjának* (serialization). Az XSL stíluslapban szereplő `xsl:output` parancs a szérializációs folyamatot szabályozza. Az XSLT 1.0 háromféle kiírási metódust támogat:

- `<xsl:output method="xml"/>`: a csomópontok jól-formázott XML formátumban kerülnek kiírásra
- `<xsl:output method="html"/>`: a csomópontok HTML 4.0-kompatibilis formátumban kerülnek kiírásra, így például az önlezáró `<br/>` címke helyett `<br>` jelenik meg a kimeneten.
- `<xsl:output method="text"/>`: csak a csomópontok értéke kerül kiírásra, jelölés nélkül

A példánkban a html metódust választottuk, így a transzformáció kimenete az alábbi lesz:

```
<html>
  <body>
    <table>
      <tr>
        <td>963 211 773 5</td>
        <td>Douglas Adams</td>
        <td>The Hitch Hiker's Guide to the Galaxy</td>
      </tr>
      <tr>
        <td>963 10 9436 7</td>
        <td>B.W. Kernighan-R. Pike</td>
        <td>The UNIX Programming Environment</td>
      </tr>
    </table>
  </body>
</html>
```

Ha stíluslap csak egyetlen sablont tartalmaz, amely közvetlenül a gyökérelemre illeszkedik, akkor lehetőség van egy **egyszerűsített írásmód** használatára is. Ekkor az `<xsl:stylesheet>` és az `<xsl:template>` elemek elmaradnak, és a sablonon belüli gyökérelem válik a stíluslap gyökerévé, az XSL névtér deklaráció pedig az új gyökérbe kerül. Így a példa stíluslap egyszerűsített írásmóddal:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <body>
    ...
  </body>
</html>
```

Az alapértelmezett kimeneti metódus az általános írásmódnál „xml”, míg az egyszerűsített írásmódnál, ha a gyökérelem `<html>`, a „html”.

5. XPath kifejezések

Az *XPath* XML dokumentumok fastruktúrájú reprezentációjában csomópontok megcímzésére szolgáló szabványos nyelv. Egy XPath kifejezés eredménye lehet csomópontok (részfák) halmaza, numerikus, szöveges vagy logikai érték.

Az előző XSL példában több ponton is használtunk már XPath kifejezéseket: az `xsl:template` parancsban a „/” kifejezés a fa gyökerét, az `xsl:for-each` parancsban a „page/ROWSET/ROW” kifejezés a sorokat reprezentáló részfákat, az `xsl:value-of` parancsban pedig az „ISBN” stb. kifejezések az aktuális részfa megfelelő mezőit reprezentáló csomópontokat választották ki.

Az XPath *elérési út* (location path) hasonló egy DOS-os vagy UNIX-os könyvtárváltó parancs útvonalkifejezéséhez: ez is egy útvonalat ad meg, csak ezúttal az XML fastruktúrában. A fájlrendszerekhez hasonlóan itt is kétféle elérési út létezik: *abszolút*, ahol az elérési út „/” karakterrel kezdődik, ekkor a kiindulási pont a dokumentum gyökere, valamit *relatív*, ahol a kiindulási pont az aktuális csomópont. Az elérési út *lépésekből* (location steps) áll, amelyeket „/” határoló választ el egymástól. Egy lépés általános formája:[26]

```
csomóponttípus[feltételes kifejezés]
```

A feltételes kifejezés rész opcionális, elmaradhat. A lehetséges csomóponttípusok a következők:

[26] Itt csak a rövidített (abbreviated) elérési út formátumot ismertetjük. A rövidítetlen (unabbreviated) formátum használatakor a lépés kiegészül az irány jelölővel.

Típus	Kiválasztott csomópont(ok)
`név`	Adott nevű gyermekcsomópontok
`@név`	Adott nevű attribútum-csomópontot
`.`	Aktuális csomópont
`..`	Szülő csomópont
`*`	Elem típusú gyermekcsomópontok
`@*`	Attribútum típusú gyermekcsomópontok
`//`	Csomópont és annak összes leszármazója
`node()`	Elem típusú gyermekcsomópontok
`text()`	Szöveges típusú gyermekcsomópontok
`comment()`	Magyarázat típusú gyermekcsomópontok

A lépésben szereplő feltételes kifejezés függvényeket és értékvizsgálatot tartalmazó összetett XPath logikai kifejezés, amelyekkel a lépésben kiválasztott csomópontok köre tovább szűkíthető. Néhány tipikus kifejezés:

Kifejezés	Igaz, ha…
`elem`	létezik az `elem` elnevezésű csomópont
`elem=érték`	az `elem` nevű csomópont értéke `érték`
`@attr=érték`	az `attr` nevű attribútum értéke `érték`
`position()=n`	ez a csomópont a halmaz `n`. eleme
`count(halmaz)=n`	az XPath elérési úttal definiált csomóponthalmaz elemeinek száma `n`
`sum(halmaz)=n`	az XPath elérési úttal meghatározott csomóponthalmaz értékeinek összege `n`

A feltételes kifejezések között az `and` és `or` logikai műveletek használhatók. A `[position()=n]` alak helyett használható az egyszerűsített `[n]` forma is. Az XPath kifejezésekben használható fontosabb függvényeket a Függelék tartalmazza.

Két XPath kifejezés által kijelölt csomóponthalmazok uniója a „|” operátorral képezhető.

Néhány XPath példa a könnyebb érthetőség kedvéért, amelyeket a korábbi kanonikus XML kimenetre értelmezünk:

- `/` : a dokumentum gyökere (nem azonos a gyökér címkével!)
- `/page/ROWSET/ROW` : az összes lekérdezés összes sorát reprezentáló részfák
- `/page/ROWSET[1]/ROW[1]/*` : az első lekérdezés első sorának mezői
- `/page/ROWSET/ROW[AUTHOR]` : azon sorok, melyekben az AUTHOR mező nem NULL értékű (azaz melyekben létezik az AUTHOR-t reprezentáló csomópont)
- `/page/ROWSET/ROW[@num > 1 and @num < 4]` : a lekérdezések második és harmadik sorait reprezentáló részfák (felhasználva a num attribútumot a kanonikus kimenetben)
- `/page/ROWSET/ROW[last()]` : a lekérdezések utolsó sorát reprezentáló részfa
- `//ROW[contains(TITLE,'Galaxy')]` : az összes olyan sor, amelynek a TITLE mezője tartalmazza az „Galaxy” stringet.
- `count(//AUTHOR)`: az AUTHOR nevű csomópontok száma
- `name(/page/ROWSET[1]/ROW[1]/*)`: az első lekérdezés gyerekcsomópontjainak neve
- `/page/ROWSET[1]/ROW | /page/ROWSET[2]/ROW`: az első és a második lekérdezés sorait reprezentáló csomópontok uniója

Emlékeztető: ha egy XPath kifejezést XSL stíluslapon használunk, akkor az XML szintaktikai előírásai miatt a foglalt karakterek (`<`, `&`) helyett a megfelelő helyettesítő szekvenciákat kell alkalmazni.

6. XSL stíluslapok hozzárendelése XML dokumentumokhoz

Egy XML dokumentumhoz tartozó XSL stíluslapot az `xml-stylesheet` feldolgozási utasítás segítségével lehet megadni a forrásdokumentumban. Az utasítás `type` attribútumának értéke kötelezően „`text/xsl`”, a stíluslap elérhetőségét pedig a `href` attribútumban kell megadni:

```
<?xml-stylesheet type="text/xsl" href="stíluslap.xsl"?>
```

A választott XML formátumú megjelenítésben minden könyvhöz egy `<book>` címke tartozik, melynek attribútumaként jelenik meg a könyv ISBN azonosítója, és értékeként a könyv legkedvezőbb ára:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <pricelist>
    <xsl:for-each select="ROWSET/ROW">
      <book isbn="{ISBN}">
         <xsl:value-of select="PRICE"/>
       </book>
    </xsl:for-each>
  </pricelist>
</xsl:template>
</xsl:stylesheet>
```

A példából az is látható, hogy egy **attribútum értékét** a `<címke attribútum="{XPath kifejezés}">` formában lehet a stíluslapon beállítani.
A szöveges megjelenítésben minden könyvhöz tartozik egy sor, amelyben az ISBN-t és az árat tüntetjük fel:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
  <xsl:for-each select="ROWSET/ROW">
    <xsl:value-of select="ISBN"/>
    <xsl:value-of select="PRICE"/>
    <xsl:text>&#10;</xsl:text>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

A 7. sorban szereplő
 szekvencia az újsor karakternek felel meg. Mivel a stíluslapban szereplő szöveges csomópontok elejéről és végéről az XSL transzformáció levágja a whitespace karaktereket, ezért használjuk az `xsl:text` utasítást. Az `xsl:text` utasítással jelölt szöveges érték változtatás nélkül kerül a kimenetre.

7. Elágazások és változók XSL stíluslapokon

7.1. Feltételes végrehajtás

```
<xsl:if test="XPath kif.">
  XSLT részlet
</xsl:if>
```

Ha igaz a `test` attribútumban megadott XPath kifejezés, akkor végrehajtja a beágyazott XSLT részletet. Amennyiben az XPath kifejezés nem logikai értéket, hanem XML részfahalmazt ad eredményül, akkor az értéke pontosan akkor igaz, ha a visszaadott részfahalmaz nem üres. Így az `<xsl:if>` (és az alább ismertetett `<xsl:choose>`) utasítás alkalmas annak eldöntésére is, hogy egy adott csomópont létezik-e az XML dokumentumon belül.

7.2. Feltételes elágazás

```
<xsl:choose>
  <xsl:when test="XPath kif.">
    XSLT részlet
  </xsl:when>
  <xsl:when test="XPath kif.">
    XSLT részlet
  </xsl:when>
  ...
  <xsl:otherwise>
    XSLT részlet
  </xsl:otherwise>
</xsl:choose>
```

Az `xsl:choose` pontosan egy XSLT részletet hajt végre: azt, ahol legelőször igaz az XPath kifejezés. Ha egyik sem igaz, az `xsl:otherwise` ág kerül végrehajtásra.

7.3. Változók

Az XSL változó elnevezés csalóka, mivel az XSL változók konstansok, csak egyszer adhatunk nekik értéket. A változók érvényességi köre az az XML címke, ill. annak összes leszármazottja, ahol a változót definiáltuk. Változóhoz rendelhető skalár érték (szöveg, szám, logikai), vagy XML részfa, ahogy az alábbi két definíció mutatja:

```
<xsl:variable name="változó_név" select="XPath kif."/>
<xsl:variable name="változó_név">XML részfa</xsl:variable>
```

A definiált változók felhasználhatóak később XPath kifejezésekben, ahol a `$változó_név` formában lehet rájuk hivatkozni.[27] Például a

```
<xsl:variable name="n" select="2"/>
<xsl:value-of select="item[$n]"/>
```

parancsok a 2. `item` részfa értékét illesztik a kimenetbe.

[27] XSLT 1.0-ban az XML részfaként (result tree fragment) definiált változók szövegesen tárolódnak és csak sztringként kezelhetőek, így az XPath csomópont címzés kifejezések nem alkalmazhatóak rájuk. A kimenetre azonban részfaként is beilleszthetőek az *xsl:copy-of* függvény segítségével.

8. XSLT Sablonok

8.1. Sablonok rekurzív feldolgozása

Mint korábban említettük, egy XSLT sablon egy csomópont transzformációjának szabályait tartalmazza. A sablonokhoz tartozik egy XPath kifejezéssel megadott minta, amely meghatározza, mely csomópontokra érvényes az adott sablon. A mintát a `match` attribútumban kell megadni. Az előző példákban csak egyetlen sablont használtunk, amely közvetlenül a gyökér elemre illeszkedett, és amely a teljes fa transzformációs szabályát tartalmazta. Több sablon használatával a transzformáció rekurzív módon is végrehajtható.
Az általános XSL transzformáció menete a következő. Az XSL transzformáció elején a gyökér csomópont az egyetlen kiválasztott csomópont. Az XSLT processzor kikeresi a kiválasztott csomópontra illeszkedő sablont, és végrehajtja a sablonban definiált szabályokat. Ha a sablon olyan XSLT utasítást tartalmaz, amely további csomópontokat jelöl ki feldolgozásra (tipikusan a gyermek csomópontokat), akkor a feldolgozás az adott csomópontokat egyenként kiválasztva folytatódik.
Az XSLT processzor tartalmaz egy beépített sablont, amely biztosítja a rekurzív feldolgozást arra az esetre, ha egy elem csomópontra nincs illeszkedő explicit sablon a stíluslapon:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

Ez a sablon nem ad semmilyen kimenetet, csupán az `xsl:apply-templates` paranccsal arra utasítja a processzort, hogy a gyermekcsomópontokon folytassa a feldolgozást.
Létezik egy másik beépített sablon a szöveges és attribútum csomópontokra is. Ezen sablon egyszerűen kimásolja a csomópont értékét a kimenetre:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

Ha egy csomópontra több sablon is illeszkedik, akkor az XSLT processzor mindig a legspecifikusabb sablont fogja kiválasztani. Ökölszabályként, a `*` mintánál specifikusabb a `VALAMI` típusú minta, és ennél is specifikusabb a `VALAMI/VALAMIMÁS`, illetőleg a `VALAMI[feltétel]` típusú minta.
Az alábbi példa azt szemlélteti, hogyan tudjuk a könyvek listáját megjeleníteni HTML táblázatban rekurzív sablonokkal.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
  <html>
    <body><xsl:apply-templates/></body>
  </html>
</xsl:template>

<xsl:template match="ROWSET">
  <table><xsl:apply-templates/></table>
</xsl:template>

<xsl:template match="ROW">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="ROW/*">
  <td><xsl:apply-templates/></td>
```

```
</xsl:template>
</xsl:stylesheet>
```

8.2. Többször felhasználható nevesített sablonok

Az XSLT lehetőséget biztosít arra, hogy a több sablonban is használt XSLT részleteket külön blokkokba, úgynevezett *nevesített sablonokba* helyezzük. A normál sablonokkal szemben a nevesített sablonokat nem mintaillesztéssel választja ki a processzor, hanem más sablonokból az az explicit `<xsl:call-template name="sablonnév">` utasítással lehet meghívni őket. A nevesített sablonok nevét az `xsl:template name` attribútumában kell megadni.
Az alábbi példa egyúttal azt is szemlélteti, hogyan adható át paraméter a meghívott sablonnak:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template name="nevesitett_sablon">
  <xsl:param name="par1" select="2"/>
  <!-- XSLT utasítások -->
</xsl:template>

<xsl:template match="ROW">
  <xsl:call-template name="nevesitett_sablon">
    <xsl:with-param name="par1" select="42"/>
  </xsl:call-template>
</xsl:template>

</xsl:stylesheet>
```

8.3. Stíluslapok tagolása

A struktúráltság végett lehetőség van egy stíluslap több fájlra tagolására, illetőleg külső stíluslapokban tárolt sablonkönyvtárak felhasználására. Külső stíluslap beillesztésére az `xsl:include` és az `xsl:import` utasítások használhatóak, mindkét utasítás az `xsl:stylesheet` közvetlen gyermekelemeként adandó meg. A beillesztett sablonok precedenciája `xsl:include` esetén azonos, `xsl:import` esetén alacsonyabb lesz, mint a hívó stíluslap sablonjaié.

9. Felhasznált irodalom

1. T. Bray et al. (editors): Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 2004.
2. J. Clark (editor): XSL Transformations (XSLT) Version 1.0, W3C Recommendation.
3. J. Clark et al. (editors): XML Path Language (XPath) Version 1.0, W3C Recommendation.
4. Oracle 10g: XML Developer's Kit Programmer's Guide, Oracle Co., 2004.
5. S. Muench: Building Oracle XML Applications, Chapter 7: Transforming XML with XSLT.
6. Oracle® XML Developer's Kit Programmer's Guide – 11g Release 2 (11.2), elérhető online: http://docs.oracle.com/cd/E11882_01/appdev.112/e23582/adx_overview.htm

10. Függelék: XPath Függvény Referencia

Csomópont halmaz függvények	
`last()`	Az aktuális csomóponthalmaz elemeinek számát adja vissza
`position()`	Az aktuális elem indexét adja vissza.
`count(node-set)`	A paraméter csomóponthalmaz elemeinek számát adja vissza.
`namespace-uri(node-set)`	Az első csomópont címkéjének névterét adja vissza
`name(node-set)`	Az első csomópont címkéjének nevét adja vissza.
Sztringkezelő függvények	
`string(object)`	Az argumentumként megadott értéket szöveges értékké konvertálja.
`concat(string,string,...)`	A paraméterként adott sztringeket összefűzi
`starts-with(string,string)`	Igaz, ha a paraméterként megadott első sztring a másodikkal kezdődik.
`contains(string,string)`	Igaz, ha a paraméterként megadott első sztring tartalmazza a másodikat.
`substring-before(string,string)`	Visszaadja az első sztring mindazon részét, amely a második sztring első sztringben való első előfordulása előtt van.
`string-length(string?)`	A megadott sztring hosszát adja vissza. Ha az argumentum elmarad, az aktuális elem értékének hosszát adja vissza.
`normalize-space(string?)`	Levágja a megadott sztring elején és végén levő whitespace karaktereket, a sztring közepén levőket pedig egyetlen szóközre cseréli. Ha az argumentum elmarad, az aktuális elem értékére végzi a műveletet.
`translate(string,string,string)`	Az első sztringben előforduló, második sztringben megadott karaktereket a harmadik sztringben megadott karakterekre cseréli. Például translate(„PHP”,”PH”,”DT”) kimenete „DTD”.
Logikai függvények	
`boolean(object)`	Az argumentumként megadott értéket logikai értékké konvertálja.
`not(boolean)`	Logikai érték ellentettjét adja vissza.
`true()`	Logikai igaz
`false()`	Logikai hamis
Numerikus függvények	
`number(object)`	Az argumentumként megadott értéket numerikus értékké konvertálja.
`sum(node-set)`	A csomóponthalmaz értkeinek összegét adja
`floor(number)`	Az argumentumot lefelé kerekíti
`ceiling(number)`	Az argumentumot felfelé kerekíti
`round(number)`	Az argumentumot kerekíti

11. Függelék: XSLT Referencia

xsl:apply-templates	
Végrehajtja az XSL transzformációt a select attribútumában megadott csomópontokra, ennek hiányában a gyermekcsomópontokra.	
`select`	Csomópontokat meghatározó XPath kifejezés
`mode`	Ha definiált, akkor ezen módra definiált sablonokkal hajtja végre a transzformációt.

xsl:attribute	
Egy attribútum csomópontot hoz létre az eredményfában. Az attribútum értékét a beágyazott szöveg definiálja.	
`name`	Az attribútum neve
xsl:call-template	
Meghív egy nevesített sablont. Paramétereket az opcionális xsl:with-param gyermekelemekkel lehet átadni.	
`name`	A nevesített sablon neve.
xsl:choose	
Elágazást definiál, az ágakat az xsl:when és xsl:otherwise gyermekelemek tartalmazzák.	
xsl:copy-of	
A kiválasztott csomópontokat átmásolja az eredményfába. Működése hasonló a value-of parancséhoz, de megtartja az eredeti fastruktúrát, nem konvertál szöveges csomóponttá.	
select	A csomóponthalmazt meghatározó XPath kifejezés
xsl:for-each	
Iterál a megadott csomóponthalmazon. Az iteráció sorrendjét az opcionális xsl:sort gyermekelem definiálja.	
`select`	A csomóponthalmazt meghatározó XPath kifejezés
xsl:if	
Ha a megadott feltétel igaz, végrehajtja a beágyazott XSLT stíluslap részletet.	
`test`	Feltételt definiáló XPath kifejezés
xsl:import	
Egy külső stíluslapot illeszt az aktuális stíluslapba, így a külső stíluslap sablonjai is elérhetővé válnak. Eltérően az xsl:include-tól, az importált stíluslap sablonjainak alacsonyabb a prioritása, mint az importáló stíluslap sablonjainak. Az xsl:import csak az xsl:stylesheet közvetlen gyermekeként adható meg, és minden más gyermeket megelőzően kell megadni.	
`href`	A külső stíluslap elérhetősége.
xsl:include	
Egy külső stíluslapot illeszt az aktuális stíluslapba, így a külső stíluslap sablonjai is elérhetővé válnak. Az xsl:include csak az xsl:stylesheet közvetlen gyermekeként adható meg.	
`href`	A külső stíluslap elérhetősége.
xsl:otherwise	
Az xsl:choose parancs egy ágát definiálja. Ha egyetlen xsl:when feltétel sem igaz, akkor az xsl:otherwise elembe ágyazott stíluslap részlet kerül végrehajtásra.	
xsl:output	
Eredményfa szérializációját vezérlő utasítás.	
`method`	Szérializációs metódus, lehetséges értékek: xml, html, text
`encoding`	A szérializációhoz használandó preferált kódtáblát deklarálja.
`omit-xml-declaration`	Beállítja, hogy az XML deklarációs utasítás kerüljön-e a kimenetre. Lehetséges értékek: yes, no
`indent`	Beállítja, hogy az XSLT processzor whitespace-ek hozzáadásával formázza-e a kimenetet. Lehetséges értékek: yes, no
`media-type`	A karakterfolyam média típusát (mime type) definiálja.
xsl:param	
Egy XSL paramétert definiál. Az xsl:variable-től eltérően, a változónak csak az alapértelmezett értékét definiálja, amely a sablonnak átadott paraméterekkel felülírható.	
`name`	Paraméter neve
`select`	A változó értékét meghatározó XPath kifejezés.
xsl:sort	
Az xsl:apply-templates és xsl:for-each utasítások gyermekelemeként adható meg, és ezen utasításokban a csomópontok feldolgozási sorrendjét határozza meg.	
`select`	A rendezési kulcsot meghatározó XPath kifejezés.
`data-type`	A kulcs adattípusát definiálja. Ha értéke text, a kulcsot lexikografikusan rendezi. Ha értéke number, akkor a kulcsot számmá konvertálja és numerikusan rendezi.
`order`	Rendezés iránya, lehetséges értékek: ascending, descending
`case-order`	Szöveges rendezésnél határozza meg, hogy a nagy, vagy a kis betűk

	kerüljenek előre. Lehetséges értékei: upper-first, lower-first.
xsl:stylesheet	
Stíluslap gyökéreleme.	
`version`	Értéke kötelezően 1.0
xsl:template	
Sablont definiál.	
`match`	XPath kifejezés, amely azonosítja azon csomópontokat, amelyekre a sablon érvényes. Nevesített sablonnál elmaradhat.
`name`	Nevesített sablon neve. Nevesítetlen sablonnál elmaradhat.
`priority`	Sablon prioritása. Ha több, ugyanannyira specifikus sablon illeszkedik egy csomópontra, a prioritás dönt, melyik kerül végrehajtásra.
`mode`	Opcionálisan megadható mód.
xsl:text	
A megjelölt szöveget változtatás nélkül kimásolja a kimenetre.	
xsl:value-of	
Szöveges csomópontot állít elő az eredményfában.	
`select`	Az értéket meghatározó XPath kifejezés. Ha a kifejezés csomóponthalmazzal tér vissza, a hozzájuk tartozó értékek kerülnek a kimenetre.
xsl:variable	
Egy XSL változót definiál.	
`name`	Változó neve
`select`	A változó értékét meghatározó XPath kifejezés.
xsl:when	
Az xsl:choose parancs egy ágát definiálja. Ha a megadott kifejezés igaz, akkor végrehajtja a beágyazott stíluslap részletet.	
`test`	A feltételt definiáló XPath kifejezés.
xsl:with-param	
Ezen utasítás az xsl:apply-templates és az xsl:call-template parancsok gyermekelemeként adható meg, és a hívott sablonnak átadott paraméter(eke)t definiálja.	
`name`	Paraméter neve
`select`	Paraméter értékét meghatározó XPath kifejezés

VI. labor: Szerveroldali PHP webalkalmazás fejlesztése Smarty sablonkezelővel

Szerző: Hunyadi Levente, Nagy Gábor, Erős Levente[28]

1. Bevezetés

Napjainkban a szerveroldalon dinamikusan generált tartalmú weboldalak széleskörűen elterjedtek. Tipikus az is, hogy az efféle tartalom hátterében adatbázis-kezelő áll. A PHP – ami egyes vélemények szerint eredetileg a Personal Home Page rövidítése – egy általános célú, ám elsősorban webes fejlesztésekre specializálódó, ingyenesen hozzáférhető parancsnyelv. A PHP számos platformon elérhető (köztük Linux, Windows, Mac OS X és számos más Unix-variáns alatt), könnyen elsajátítható és szervesen beépül egy-egy weboldal jelölőnyelvi kódjába: mindezek ideálissá teszik webes fejlesztésekhez. A nyelv számos eszközt biztosít a szövegfeldolgozásra a reguláris kifejezésektől az XML dokumentumok

[28] A segédlet Laczay Bálint, Salamon Gábor és Surányi Gábor *PHP: Hypertext Preprocessor* című munkája alapján készült.

feldolgozásáig. Függvénykönyvtárai révén azonban olyan összetett feladatok ellátására is alkalmas, mint képek generálása, PDF dokumentumok előállítása vagy állományok tömörítése.
Egy PHP parancsállomány (szkript) működésének lépései a következők:

1. A felhasználó böngészője egy konvencionálisan `php` kiterjesztésű oldalt megcímző HTTP kérést küld a webkiszolgálóhoz.
2. A kiszolgáló ennek hatására – megfelelő beállítások esetén – a PHP értelmezőnek adja át a vezérlést.
3. Az értelmező PHP szkriptként feldolgozza az oldalt, és a szkript kimenetét a webkiszolgálónak adja vissza.
4. A webkiszolgáló ezt a kimenetet visszaküldi a felhasználó böngészőjének.

Az adatbázis-elérés PHP-ben a már említett függvénykönyvtárakon keresztül történik. A legtöbb adatbázis-kezelőhöz létezik függvénykönyvtár, ezeket natív függvénykönyvtáraknak hívjuk. Sajnos ezek adatbázis-kezelőnként eltérő felépítésűek, használatuk módja más és más. Van azonban egységes adatbázis-elérést lehetővé tevő függvénykönyvtár is, a szabványos ODBC (Open Database Connectivity) felület. Az egységes felület hátránya, hogy kevésbé hatékony, mint a natív elérés és az egyes adatbázis-kezelők speciális lehetőségeit sem képes kihasználni.
A labor során Oracle adatbázis-kiszolgálóhoz fogunk csatlakozni az Oracle8 natív felületen át.[29] E felületet a PHP-értelmező az OCI8 (Oracle Call Interface) függvénykönyvtár segítségével nyújtja, tehát a webkiszolgálóra annak telepítése is szükséges.
Webes fejlesztéseknél is célszerű szem előtt tartanunk a korábbról már ismert modell–nézet–vezérlő (model–view–controller, MVC) paradigmát. Jelen labor folyamán inkább *modellről* és egymástól el nem különülő *nézet-vezérlőről* fogunk beszélni, ugyanis a böngésző alkalmazásban megjelenített felület vezérlésével nem kell a fejlesztés során törődnünk. Az ilyen típusú fejlesztések megkönnyítéséért alakultak ki a sablonkezelő rendszerek (template engines), mint például a Smarty. A választás azért esett épp a Smarty-ra, mert egyszerűsége miatt könnyen tanulható, és kellően elterjedt ahhoz, hogy a későbbiekre nézve is hasznos lehessen a labor során megszerzett tudás.
Mivel a labor során az adatbázis-használat egy újabb aspektusának megismeréséhez egy új nyelv és egy sablonkezelő rendszer elsajátítása is szükséges, a feladat elkészítését egy példaalkalmazás illusztrálja. A forrásszinten is közzétett példaalkalmazás a tárgy honlapjáról érhető el.

2. PHP alapismeretek

A következő néhány szakaszban egy rövid bepillantást nyújtunk a PHP nyelvbe. A cél egy olyan minimális ismeretanyag átadása, amely egyrészt elegendő a labor elején a beugró PHP-t érintő részeinek megválaszolásához, másrészt alapul szolgál a teljes PHP dokumentáció hatékony megértéséhez. A http://www.php.net/manual/en/ oldal ismerteti teljes körűen a nyelv képességeit és a konkrét függvénykönyvtárak tartalmát.[30] Mindazonáltal a fejezet végén található példaprogram megértése után a mérés teljesítése nem jelenthet problémát.
A bemutatásuk nélkül támaszkodunk az (X)HTML és a C nyelv alapszintű ismeretére, mivel e segédlet keretét meghaladná bemutatásuk. A HTML és az XHTML alapfogalmaival és a webes űrlapkezeléssel kapcsolatban a http://www.w3.org/TR/html4, ill. a http://www.w3.org/TR/xhtml1 oldalakon elérhető dokumentációk előzetes tanulmányozását ajánljuk.

[29] Létezik még az Oracle nevű függvénykönyvtár is, ami az Oracle8 sokkal szerényebb képességekkel rendelkező elődje.
[30] A magyar nyelvű változat a http://www.php.net/manual/hu/ címen található.

Az XHTML és a HTML nyelv szinte azonos, gépi feldolgozhatóság és áttekinthetőség szempontjából azonban az előbbi előnyösebb, mivel (XML értelemben) jól formált, azaz érvényesek rá az XML megkötései. A kettő közötti fontosabb különbségeket itt is megemlítjük:

- Míg a HTML nyelvben az átfedő elemek (angol nevén tag) használata a gyakorlatban gyakran tolerált, az XHTML-ben nem. Például a `<b>félkövér <i>félkövér dőlt</b> dőlt</i>` kódrészlet helyett XHTML-ben a `<b>félkövér <i>félkövér dőlt</i></b><i> dőlt</i>` használandó.
- Az önlezáró elemek, mint a `<br>`, XHTML-ben nem legálisak, például `<br>` helyett a `<br/>` használatos.[31]
- Az XHTML elemeket mindig le kell zárni (`<p>` után például `</p>` kell, hogy következzen valahol a kódban).
- Az XHTML elemek nyitó és záró címkéi kisbetűkkel írandók (`<HTML>` helyett `<html>`).
- Az XHTML elemek attribútumainak értékét kötelező egyszeres vagy kétszeres idézőjelek közé tenni (`href="1.html"` vagy `href='1.html'`).

2.1. PHP szkriptek

A PHP oldalak szöveges dokumentumok, és a hagyományos HTML oldalakhoz hasonló módon kell őket a webkiszolgálón elhelyezni. A `php` kiterjesztés hatására a kiszolgáló ezeket PHP oldalakként fogja értelmezni.[32]
A PHP oldal HTML és (végrehajtandó) szkript részek váltakozó sorozatából áll.[33]
A HTML részek külön jelölése nem szükséges, ez az alapértelmezés. Tetszőleges jelölőnyelvi szöveget helyezhetünk el bennük, ez betűről betűre bele fog kerülni a böngészőnek visszaküldött válaszba. A PHP tetszőlegesen kis- és nagybetűket tartalmazó utasításait tartalmazó szkript részeket `<?php` és `?>` jelek közé kell zárni. Az utasítások közötti elválasztójel – a PERL és a C nyelvvel való rokonság miatt – a `;` (pontosvessző). Megjegyzéseket írhatunk a `/*` és `*/` jelek közé, illetve `//` vagy `#` után a sor végéig. Az utasításblokkokat `{` vezeti be és `}` zárja le.
Elöljáróban egy példa PHP oldalt mutatunk be, ami a hetes szorzótáblát írja ki:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Szorzótábla</title>
</head>
<body>
  <table>
    <tr><th>x</th><th>7*x</th></tr>
    <?php
      for ($i = 1; $i <= 10; $i++) {
        print "<tr><td>";
```

[31] XHTML-ben ugyan alkalmazható az önlezáró tagek ekvivalens hosszú formája, azaz pl. `<br></br>` is, ez azonban olyan böngészők számára, amelyek nem képesek XHTML, csak HTML elemzésére, zavart okoz. Ezért bevett gyakorlat a rövid alak használata, a segédletben is ezt alkalmazzuk.
[32] A legtöbb kiszolgáló-alkalmazásban szabadon megadható, hogy mely fájlokat kapja meg a PHP értelmező (interpreter) futtatásra, így ez pl. eltérő kiterjesztés hozzárendelésével vagy más szabály alapján is meghatározható.
[33] A PHP oldalak (és más szöveges dokumentumok, pl. XHTML) szerkesztéséhez egy könnyen és sokrétűen használható eszköz a Notepad++ (http://notepad-plus.sourceforge.net/). A labor feladatainak elkészítéséhez mindenképpen ajánljuk valamilyen PHP- és XHTML-szerkesztést támogató szoftver használatát.

```
        print $i;
        print "</td><td>";
        print 7*$i;
        print "</td></tr>\n";
      }
    ?>
  </table>
</body>
</html>
```

2.2. Típusok

A PHP-ben leggyakrabban használt típusok a logikai érték, az egész és a lebegőpontos szám, a füzér (karaktersorozat) és a tömb.[34] Logikai értékeket a `TRUE` és `FALSE` (vagy `true` és `false`) konstansokkal, számokat legegyszerűbben tízes számrendszerbeli alakjukkal, míg karaktersorozatokat aposztrófok (`'`) vagy idézőjelek (`"`) között adhatunk meg. Aposztrófok használatakor a füzéren az értelmező semmilyen változtatást sem végez,[35] míg idézőjelek alkalmazásakor a változóhivatkozásokat (amelyeket `$` jelöl) behelyettesíti és néhány escape-szekvenciát is értelmez. Így

- `$a = '\n'` hatására `$a` értéke a visszafelé döntött törtvonal–n karakterpáros lesz, és nem a sortörés,
- majd `$b = "{$a}\n{$a}"` hatására b értéke két visszafelé döntött törtvonal–n karakterpáros lesz (változó értékének helyettesítése) egy sortöréssel elválasztva (escape-szekvencia értelmezése).

Egy változó, illetve érték típusát (szinte) sosem adjuk meg explicit módon, ez mindig értelmezéskor dől el, méghozzá az alapján a környezet alapján, amiben használjuk őket, szükség esetén a típuskonverzió automatikusan lezajlik. Az alábbiakban a leggyakrabban előforduló konverziókat soroljuk fel:

- Számból karaktersorozat: a karaktersorozat a szám tízes számrendszerbeli alakja lesz, nagy számok esetén exponenciális jelöléssel. Például `134.7` karaktersorozattá konvertálva `"134.7"` lesz, `10000000*10000000` pedig `"1E+014"`.
- Karaktersorozatból szám: az `strtod()` C hívásnak megfelelően, a karaktersorozat elején található szám lesz az érték, például `"3 kismalac"` értéke számként `3`, `"nekem 8"` értéke pedig `0`, mivel a karaktersorozat elején nincs szám.
- Logikai értékből karaktersorozat: `FALSE` illetve `TRUE` a logikai értéknek megfelelően üres karaktersorozatot vagy `"1"` értéket ad.
- Karaktersorozatból logikai érték: `""` és `"0"` logikai értékként `FALSE`, minden más karaktersoroat `TRUE`. Tehát `"true"` és `"false"` értéke is `TRUE`.
- Számból logikai érték: `0` és `0.0` logikai értékként `FALSE`, minden más szám `TRUE`.
- Logikai értékből szám: `FALSE` számként `0`, `TRUE` számként `1`.
- Tömbből logikai érték: üres tömb logikai értékként `FALSE`, legalább egy elemet tartalmazó tömb `TRUE`.

Természetesen lehetőség van explicit típuskonverzióra is, ezt a C nyelvből ismert cast operátorral tehetjük meg: `$i = (int) 1.7` értéke `1` vagy `$s = (string) 2.5` értéke `"2.5"`

2.3. Operátorok

A PHP legfontosabb operátorai:

- Aritmetikai operátorok: `+` (összeadás), `-` (kivonás), `*` (szorzás), `/` (lebegőpontos osztás), `%` (maradékképzés), egyoperandusú `-` (negálás): ezek számokat várnak paramétereikként.

[34] A PHP ezen kívül három további típusfajtát ismer, amelyek az objektum (az objektum-orientált nyelvekből ismert osztályfogalom itt is ismert, az objektum típus a különféle osztálypéldányok összefoglaló neve), az erőforrás (amely külső erőforrásokra való referencia (pl. egy adatbázis-kapcsolat) tárolására szolgál), illetve a NULL típus (amelynek egyetlen példánya az SQL laborról már ismert speciális NULL érték).

[35] Ha a füzérben aposztrófot szeretnénk szerepeltetni, azt a `\'` szekvenciával tehetjük meg. Hasonlóképpen, `\` beillesztéséhez a `\\` szekvenciát kell használni. Más karaktereknek azonban nincs speciális jelentése.

Léteznek a C-ből megszokott, változókon értelmezett `++` és `--` pre- és poszt-inkrementáló illetve dekrementáló operátorok is. A C nyelvből ismert egész osztást maradékképzés és kivonás segítségével vagy kerekítéssel lehet megvalósítani.

- Értékadás, jele: `=`. A bal oldalán változóhivatkozásnak kell állnia. Létezik (a C-ben is megtalálható) más operátorokkal összevont alakja is: `+=`, `-=` stb.
- Összehasonlító operátorok, jeleik `==`, `!=`, `<`, `<=`, `>`, `>=`. Ha az operandusok típusa eltér, konverzió történik. Ha valamelyik operandus logikai érték, akkor az értelmező mindkét operandust logikai értékké konvertálja. Szám és füzér összehasonlításakor a PHP a füzért számmá alakítja; hasonlóképpen jár el két számot tartalmazó füzér esetén is. Két karaktersorozat összevetése a tárolt értéktől függően így numerikus vagy lexikai összehasonlítással is történhet.
- Típushelyes összehasonlító operátor, jele `===`. Működése az `==` operátoréhoz hasonló, csak nem végez előzetes típusegyeztetést: ha nem azonos típusú a két operandusa, `FALSE` értékkel tér vissza. A `===` negálása a `!==` operátor.
- Karaktersorozat-összefűző operátor, jele `.` (pont). Két karaktersorozat operandust vár, és ezek konkatenáltját adja eredményül. `$a . $b` hatása megegyezik a `"{$a}{$b}"` jelöléssel.

Példák az operátorok használatára:

- `4/8` értéke `0.5` (szám).
- `4/"6ökör"` értéke `0.666667` (szám), és `5 + "3 kismalac"` értéke `8` (szám), mert a `/` és a `+` operátor számmá konvertálja operandusait.
- `"FALSE" != FALSE` értéke `TRUE` (logikai érték), hiszen a `!=` operátor operandusai között van logikai érték, ezért az értelmező mindkettőt logikai értékké alakítja.
- `(10 * "1 kiskutya") . "1 kiskutya"` értéke `"101 kiskutya"`.

2.4. Változók

A változókat egy alfanumerikus karaktereket és alulvonást tartalmazó névvel azonosíthatjuk, azzal a korlátozással, hogy numerikus jeggyel nem kezdődhet. A változókra történő hivatkozás mindig `$` (dollár) jel segítségével történik, a változók nevében a kis- és nagybetűk megkülönböztetésével. Így például az `$alma_1` és az `$Alma_1` változónevek különböző változókat jelölnek. A változók deklaráció nélkül használhatóak, típusuk értékadáskor dől el. Például `$b = 4` után a `$b` nevű változó szám, majd `$b = $b . "2 a válasz"` hatására a `$b` változó karaktersorozat lesz, értéke `"42 a válasz"`. Ekkor a `$b + 3` kifejezés értéke `45`, mert ilyenkor `$b` értéke ismét számmá válik. Fontos megjegyezni, hogy a konverzió csak az operandusok kiértékelésénél játszik szerepet, a változó típusát *nem* változtatja meg. Tehát `$b` továbbra is `"42 a válasz"` marad.

2.5. Tömbök

A PHP tömbjei asszociatívak, ez azt jelenti, hogy indexük tetszőleges érték (nem csak nemnegatív egész szám) lehet. Tömböt legegyszerűbben az `array` kulcsszóval lehet létrehozni. Például: `$gyümölcs = array("ananász", "avokádó", "narancs")`. Ennek hatására a `$gyümölcs` tömb `0` indexéhez az `"ananász"`, `1` indexéhez az `"avokádó"`, `2` indexéhez pedig a `"narancs"` karaktersorozat fog tartozni. Ha nem számokat akarunk indexként használni, akkor írhatjuk például a következőt:

```
$zöldség = array(
  "s"=>"sárgarépa", "f"=>"fehérrépa", "c"=>"cukorrépa"
);
```

Többdimenziós tömbök is használhatók, például:

```
$adat = array(
  "gyümölcsök" => array("ananász", "narancs", "szilva"),
  "színek" => array("sárga", "narancs", "kék")
```

```
);
```

Egy tömb elemeit szögletes zárójelek közé tett indexértékekkel érhetjük el, így az alábbi kódrészlet értéke `"narancs"` lesz:

```
$adat["gyümölcsök"][1]
```

Ilyen módon új elemeket is elhelyezhetünk a tömbbe, például:

```
$zöldség["z"]="zöldborsó"
```

vagy csak egyszerűen:

```
$gyümölcs[] = "ribizli"
```

ilyenkor a PHP automatikusan generálja az indexet a tömbben lévő legnagyobb numerikus index alapján.

2.6. Írás a kimenetre

Viszonylag gyakori, hogy egyszerű szöveget szeretnénk beilleszteni a böngészőnek elküldött oldalba. Ekkor meglehetősen rosszul olvasható kódot kapunk a `<?php ... ?>` jelölők folyamatos használatával. Erre mutatunk egy példát az alábbiakban. (A példa alapján sejthető: a `getdate` függvény egy asszociatív tömböt ad vissza, aminek `"hours"` indexű eleme azt tartalmazza, hogy éppen hány óra van.)

```
<?php $date = getdate(); ?>
<p>Üdvözöljük, jó
<?php if ($date["hours"]<12) { ?>reggelt<?php } else { ?>napot<?php } ?>
!</p>
```

Másik lehetőség a közvetlen kiírásra, hogy a PHP `print` vagy `echo` utasítását fogjuk munkára. A kulcsszó után egy kifejezés adandó meg, amit az utasítás karaktersorozattá konvertál. Ennek használatával a fenti példa:

```
<?php $date = getdate(); ?>
<p>Üdvözöljük, jó
  <?php
    if ($date["hours"] < 12) {
      echo("reggelt");
    } else {
      echo("napot");
    }
  ?>!
</p>
```

2.7. Vezérlési szerkezetek

A PHP vezérlési szerkezetei is ismerősek lehetnek más programozási nyelvekből. Létezik `if ... else ...`, `while`, `do ... while`, `for`, `break`, `continue`, `switch` szerkezet. Ezek használata teljesen megegyezik például a C nyelvben megszokottal. Bevezették továbbá az `elseif`-et, ami működésében azonos az `else if` sorozattal.

A `foreach` szerkezet segítségével tömbök első dimenziójában tárolt értékeken lehet elemenként egyszerűen végigmenni. Használatakor egy tömbbé kiértékelődő kifejezést és egy ill. kettő változót kell megadnunk. Példaként tekintsük az alábbi egysoros kódrészletet:

```
foreach ($gyümölcs as $név) echo("Szeretem a(z) {$név}-t<br />\n");
```

és kimenetét:

```
Szeretem a(z) ananász-t<br />
Szeretem a(z) avokadó-t<br />
Szeretem a(z) narancs-t<br />
```

```
Szeretem a(z) ribizli-t<br />
```

A következő kódrészlet a `normal` tömböt másolja át `reversed`-be, úgy, hogy a kulcsokat és az értékeket felcseréli. (Természetesen csak akkor működik pontosan így, ha az eredeti értékek a tömbben egyediek.)

```
foreach ($normal as $key => $value) $reversed[$value] = $key;
```

2.8. Külső állományok hivatkozása

A `require` és az `include` kulcsszók egyaránt külső fájlok behívására szolgálnak. Az előbbi pontosan úgy működik, mint a C preprocesszora, tehát még az aktuális fájl végrehajtása előtt kifejtődik, míg az utóbbinál a kifejtés elhalasztódik addig, amíg az utasításra nem kerül a vezérlés. Emiatt lehetséges `include`-nál változókat is használni. Ami meglepő lehet – ezért hívjuk itt fel rá a figyelmet – hogy a beszerkesztett fájlokban a PHP utasításokat szintén a már ismertetett `<?php ?>` jelölőbe kell zárni, függetlenül attól, hogy a hívás helyén éppen PHP kontextusban vagyunk, különben az értelmező nem dolgozza fel azokat.

2.9. Függvénydefiníciók

A függvények definícióját a `function` kulcsszóval kell kezdeni. Ezt követi a függvény neve, majd a formális paraméterek listája. A típusokról korábban elmondottak alapján a visszatérési érték és a paraméterek típusainak megadására nincs szükség. A paraméterek neveit a változókhoz hasonlóan `$`-nak (dollárjelnek) kell megelőznie. A PHP támogatja nemcsak az érték, hanem a referencia szerinti paraméterátadást is. Ehhez a függvénydefinícióban a megfelelő paraméterek dollárjele elé `&` (et vagy és) jelet kell tenni. Visszatérésre a `return` kulcsszó szolgál, amely után megadható a visszatérési értéket előállító kifejezés is. Egyszerre több értéket visszaadni így tömbök vagy kimeneti argumentumok segítségével lehet. Például:

```
function muveletek($a, $b, &$c) {
  $c = $a + $b;
  $d = $a - $b;
  return array($c, $d);
}
```

A PHP szkript főprogramját a függvénytörzseken kívül található utasítások alkotják. A függvényeken kívül definiált változók (alapértelmezésben) a függvényeken belül nem láthatók. Amennyiben mégis szükség van valamelyikükre, azokat paraméterként adhatjuk át vagy importálhatjuk a függvény törzsébe a `global` kulcsszóval.

2.10. Előredefiniált változók

A PHP rendkívül sok előredefiniált változót bocsát a programozó rendelkezésére. Jó néhány közülük a rendszer és a szkript adatait tartalmazza, de számunkra mégis azok a legfontosabbak, amelyek a weboldalak közötti kommunikációt teszik lehetővé. Ez utóbbiaknak három fajtája van:

1. a kérés URL-jében explicite átadott paraméterek értékei
2. a kérés alapjául szolgáló oldal űrlapjának beviteli mezőivel kapcsolatos értékek
3. a beállított cookie-k értékei[36]

[36] A HTTP állapotmentes protokoll, ami azt jelenti, hogy a kiszolgáló az egyes kérések megválaszolása után semmiféle információt sem tárol el, amit aztán később felhasználhatna (pl. a következő válaszhoz). Adatbázis-kezelésnél és más alkalmazásoknál azonban gyakran szükség van bizonyos adatok (pl. vásárlókosár tartalma) megőrzésére az egyes kérések között. Alapvetően kétféle lehetőség kínálkozik ennek megvalósítására: a munkamenetek (session) és az ügyfél oldali alkalmazások. Előbbi esetben az adatok a szerveren tárolódnak és azokat valamilyen, a kliensen tárolt, minden egyes kéréshez hozzákapcsolt kulcs segítségével érjük el, míg utóbbi esetben a teljes információhalmaz a kliensen tárolódik, és csak az éppen szükséges adatokat kapcsoljuk a

Mi a labor során csak az első két csoportot fogjuk használni, tehát itt is azt ismertetjük.
Amikor a böngésző eljuttat egy HTTP kérést a kiszolgálóhoz, lehetősége van paraméterek átadására. Ezek a paraméterek a kiszolgálón futó alkalmazás (jelen esetben PHP szkript) számára elérhetőek, ezáltal tudja a kiszolgáló válaszát dinamikusan, a kérés paramétereitől függően kialakítani. A HTTP kérés paraméterei névből és a hozzá tartozó értékből állnak.
A HTML oldalban szereplő űrlapmezők az űrlap elküldésekor a kérés paramétereiként jelennek meg. A paraméter neve az űrlapmező `name` attribútumában, értéke pedig az űrlapmező `value` attribútumában szereplő érték, amelyet egyes űrlapmezőknél (pl. beviteli mező) a felhasználó megváltoztathat. Speciális esetet képeznek a jelölőmezők, amelyek csak akkor szerepelnek név-érték párként a kérés paraméterei között, ha az űrlap elküldésekor jelölt állapotban vannak. Hasonlóképpen, egy legördülő lista (`select` HTML elem) elemei közül csak az éppen kiválasztott(ak) értéke jelenik meg a HTTP kérés paraméterei között. Érdemes megemlíteni a `hidden` (rejtett) űrlapmezőt is, amely a böngészőben nem jelenik meg, értéke azonban a többi elemhez hasonlóan eljut a kiszolgálóhoz a kérés paramétereként. Ezáltal jól használható például azonosítók átadására.
Űrlapadatokat továbbítani GET és POST módokon lehet. A GET mód során az űrlap adatai a kérés URL-jében (az URL `?` utáni szakasza), POST esetén a kérés törzsében (payload) jelennek meg. Az űrlapból származó összes adat ennek megfelelően a globális `$_GET`, ill. `$_POST` nevű asszociatív tömbbe kerül, aminek kulcsai a weboldalon a mezők definiálásakor a `name` attribútumnál megadott értékek. Mivel a `$_GET` tömb tartalmazza a kérés URL-jében szereplő összes paraméter értékét, a kéréshez hozzáfűzött saját paramétereket is innen nyerhetjük ki. Pl. `index.php?id=82` URL esetén a `$_GET` tartalmazza majd a `82`-es értéket (szövegként) az `id` kulcs alatt.[37] A `$_GET` és a `$_POST` mellett egy harmadik tömb, a `$_REQUEST` segítségével férhetünk hozzá adatainkhoz, ha nem kívánunk foglalkozni azzal, azok GET vagy POST-jellegű forrásból származnak.
Az űrlapváltozók nevein (name) a PHP két változtatást végez. Ha egy név tömbindexelő operátort (`[]`) tartalmaz, az érték a névnek megfelelő tömb elemeként kerül a `$_GET`, illetve a `$_POST` tömbbe, így adva például lehetőséget arra, hogy egy listában több kijelölt elemet is feldolgozhassunk. Másrészt, mivel a . (pont) nem megengedett a változónevekben, azokat a PHP _-ra (aláhúzásra) cseréli.
Az előbb leírtak könnyebb megértéséhez tekintsük a következő összetett példát. Legyen egy regisztrációs weboldal (`register.html`) HTML forrása a következő:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Regisztrációs oldal</title>
</head>
<body>
<form action="register.php" method="post">
  <table>
  <tr>
    <td>Név:</td>
    <td><input type="text" name="name" /></td>
  </tr>
  <tr>
    <td>E-mail</td>
```

kéréshez. A Java kisalkalmazások (appletek) tipikus példái az ügyfél oldali alkalmazásoknak, míg munkameneteket akár PHP-ben is megvalósíthatunk.

[37] Mivel a `$_GET` ill. `$_POST` elemeire a feldolgozó oldal általában gyakran hivatkozik, mód van arra, hogy a PHP az első tömbdimenzió értékeit automatikusan azok kulcsaival megegyező nevű változókban is elérhetővé tegye. Ez azonban biztonsági kérdéseket vet fel, ami miatt a PHP újabb verzióiban (és a labor során is) alapértelmezetten a szolgáltatás kikapcsolt állapotban van.

```
    <td><input type="text" name="email" value="yourname@example.com"
/></td>
  </tr>
  <tr>
    <td>Sör:</td>
    <td>
      <select multiple="multiple" name="beer[]">
        <option value="Miller">Miller</option>
        <option value="Guinness">Guinness</option>
        <option value="Stuttgarter">Stuttgarter Schwabenbräu</option>
      </select>
    </td>
  </tr>
  <tr>
    <td><input type="submit" /></td>
  </tr>
  </table>
</form>
</body>
</html>
```

A `register.php` nevű feldolgozóoldal pedig legyen:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Sikeres regisztráció</title>
</head>
<body>
<p><?php
  print "Kedves {$name}, sikeresen regisztrált {$email} címmel.\n";
  $drinks = implode(', ', $beer);
  print "Az ön által rendelt italmárkák: {$drinks}."
?></p>
</body>
</html>
```

Ha a felhasználó a `register.html` oldalt behívva névként `Sir Arthur Conan Doyle`-t, címként `arthur@example.com`-ot ad meg, majd bejelöli a `Miller` és `Guinness` sört és elküldi az adatokat, a `register.php` szkript a következő HTML kódot fogja előállítni:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Sikeres regisztráció</title>
</head>
<body>
<p>Kedves Sir Arthur Conan Doyle, sikeresen regisztrált arthur@example.com
címmel.
Az ön által rendelt italmárkák: Miller, Guinness.</p>
</body>
</html>
```

Látható, hogy ha a felhasználó a `SELECT` űrlapmezőnek egyszerre több elemét is kiválasztja, a HTTP kérésben minden egyes elemhez generálódik egy név-érték pár (pl. `beer[]="Miller"` és `beer[]="Guinness"`). A PHP szkript, megkapva az ezeket a paramétereket tartalmazó kérést, létrehozza a `beer` tömböt, majd a `0` és az `1` sorszámú elemeit feltölti a `"Miller"`, illetve a `"Guinness"` füzérekkel. Az `implode` utasítás a paraméterként kapott tömb elemeit fűzi össze.

3. Smarty alapismeretek

Webes környezetben az MVC (model–view–controller, modell–nézet–vezérlő) paradigma szerinti programozás biztosítására legalkalmasabb eszközök a sablonkezelő rendszerek. Ebben a fejezetben a Smarty sablonkezelő rendszer alapjaiba nyújtunk bepillantást. Az itt bemutatott nyelvi elemek elegendőek a labor során elvégzendő feladatok sikeres teljesítéséhez, bővebb információ a http://www.smarty.net/ weboldalon található.

Smarty-val történő fejlesztés során egyrészt létre kell hoznunk egy PHP nyelvű alkalmazást, amely adatbázis-lekérdezések, illetve különböző számítási műveletek segítségével előállítja a megjelenítendő adatokat, ezáltal megvalósítva a *modellt*, másrészt implementálnunk kell egy XHTML sablon-dokumentumot, amely egyrészt tartalmazza a megjelenítéshez szükséges formázási elemeket, másrészt kijelöli azon pontokat, ahová a PHP alkalmazás által szolgáltatott adatok fognak helyettesítődni. A sablon tehát a *nézetet* valósítja meg. A PHP szkript és a sablon közötti kapcsolatot egy kitüntetett Smarty objektum példányosítása, a paraméterek átadása és a sablonfájl megjelenítése jelenti a PHP alkalmazás kódjának végén, a *vezérlő* funkcióját tehát a Smarty objektum és szolgáltatásai töltik be.

Tekintsünk most egy egyszerű alkalmazást, amely a fent leírt koncepciót világossá teszi (a példában használt Smarty elemek magyarázatát lásd később). Az alkalmazás az adatbázisban található személyek adatait listázza ki.

Első lépésként meg kell írnunk egy `people.php` nevű PHP szkriptet, amely lekérdezi a személyek adatait az adatbázisból. Az áttekinthetőség érdekében a hibákat ellenőrző vezérlési szerkezeteket az alábbi példában elhagytuk, de a feladatok megoldása során ezek használata természetesen elvárt.

```
<?php
require("../../php_include/config.php");
...

$stmt = oci_parse($conn, "SELECT name, birthdate, email FROM people");
oci_execute($stmt);
oci_fetch_all($stmt, &$people);

$smarty = new Smarty();
$smarty->assign("people", $people);
$smarty->display("people.tpl");
?>
```

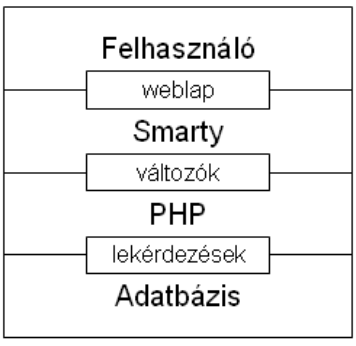
A PHP alkalmazás tehát első lépésben lekérdezi az adatbázisban tárolt személyek nevét, születési dátumát és e-mail címét, amelyet a `people` nevű tömbben tárol el (az adatbázis-elérést végző függvények leírását lásd a 4. fejezetben). Ezek után létrehoz egy `Smarty` osztályhoz tartozó objektumot `$smarty` néven, az `assign` metódushívással átadja a `$people` nevű PHP változó értékét a `people` nevű, a sablon számára elérhető változónak, végül pedig megjeleníti a `people.tpl` nevű sablonfájlt a `display` metódussal.

Nézzük a sablonfájl forrását:

```
...
<body>
{foreach item=person from=$people}

Profile of {$person[0]}:<br/>
Birthdate: {$person[1]}<br/>
E-mail: {$person[2]}<br/>

{/foreach}
</body>
```



```
...
```

A sablonfájl tehát nem más, mint egy Smarty-specifikus elemekkel felcímkézett (annotált) XHTML oldal. A fenti sablon a PHP szkript által előállított `$people` tömb elemein megy végig, és minden elemnek kiírja az első, a második és a harmadik elemét, azaz a megfelelő személy nevét, születési dátumát és e-mail címét.
A Smarty sablonkezelő, a PHP parancsértelmező és az adatbázis-kezelő együttműködését az 1. ábra szemlélteti.
A következőkben áttekintjük a Smarty lényegesebb nyelvi elemeit.

3.1. Változók és tömbök

A Smarty sablonban használt változók értékét az `assign("smartyvar",$phpvar)` metódussal adhatjuk meg a PHP kódban, ahogy azt a példaalkalmazásban már láthattuk. A változókra `{$smartyvar}` formában hivatkozhatunk a sablonban.
Asszociatív és nem asszociatív tömböket is átadhatunk a sablonnak. Asszociatív tömbök átadása esetén a PHP-hoz képest annyi a különbség, hogy a tömb elemeire `{$tomb.kulcs}` formában hivatkozhatunk.

3.2. if

Az `if` szerkezet szintaxisa Smarty-ban a következő:

`{if kifejezés}` *`egyik ág`* `{else}` *`másik ág`* `{/if}`

Amennyiben tehát az `if` kulcsszó után megadott logikai kifejezés értéke igaz, *`egyik ág`* által jelölt XHTML kódrészlet fog végrehajtódni, ha nem igaz, akkor pedig *`másik ág`* által jelölt XHTML kódrészlet. Lehetőség van az `{elseif}` kulcsszó használatára is.

3.3. foreach

A `foreach` szerkezetet használva egy tömb elemein haladhatunk végig. Szintaxisa:

```
{foreach item=book from=$books}
...
{foreachelse}
...
{/foreach}
```

A `foreach` kulcsszó után két paraméternek kell állnia. A `from` paraméter azon tömb nevét adja meg, amely elemein végig kívánunk haladni, míg az `item` paraméterben pedig azt adhatjuk meg, hogy milyen néven kívánunk hivatkozni az aktuális tömbelemre. A szerkezetben opcionális egy `{foreachelse}` kulcsszóval bevezetett szakasz, amelyben azt írjuk le, mi történjen, ha `{foreach}` nem adott vissza eredményt. A szerkezetet `{/foreach}` zárja le.

3.4. section

A `section` szerkezet a `foreach`-hez hasonlóan arra alkalmas, hogy egy tömb elemeit járjuk végig vele. Szintaxisa a következő:

```
{section name=idx loop=$arrayofitems}
...
{sectionelse}
...
{/section}
```

A `section` kulcsszót két paraméter követi. A `name` paraméter egy minden iterációban inkrementálódó kifejezés. A `loop` paraméter a bejárni kívánt tömböt tároló változó neve. Ezeken felül három lényegesebb opcionális paramétert is megemlítünk. A `start` paraméter azt határozza meg, hogy az első iterációban mi legyen a növelendő paraméter értéke, a `max` pedig azt, hogy milyen értéknél álljon le a hurok ismételt végrehajtása. A `step` paraméterrel adhatjuk meg, hogy mennyivel inkrementálódjon a `name` paraméter értéke iterációnként. Amennyiben opcionális paramétereket nem adunk meg, úgy az iterációk száma a végigjárt tömb hossza lesz. A `{sectionelse}` a `{foreachelse}` vezérlőhöz hasonlóan működik. A szerkezetet `{/section}` zárja le.

3.5. php

A `php` szerkezettel bárhol PHP kódot helyezhetünk el a sablonban. Használata:

```
{php}
PHP kód
{/php}
```

3.6. include

Az `include` utasítással más sablonokat ágyazhatunk be az adott sablonba. Szintaxisa a következő:

```
{include file='included.tpl'}
```

A beágyazni kívánt sablon nevét tehát a `file` paraméterben adjuk meg.

3.7. Logikai kifejezések

A Smarty-ban használható logikai kifejezések a következők:

Operátor	Alternatív jelölések	Jelentése
==	eq	egyenlő
!=	ne, neq	nem egyenlő
>	gt	nagyobb
<	lt	kisebb
>=	gte, ge	nagyobb vagy egyenlő
<=	lte, le	kisebb vagy egyenlő
===		típushelyes összehasonlítás
!	not	nem
%	mod	modulus
is [not] div by		[nem] osztható
is [not] even		[nem] páros
is [not] odd		[nem] páratlan

4. A fontosabb Oracle8 függvények

Ebben a szakaszban vázlatosan ismertetjük a mérés során használandó Oracle8 függvényeket. Bár a PHP kódban nem használunk típusneveket, ezen könyvtári függvények esetében mégis megadjuk, hogy bemeneti paramétereiket belső működésük során milyen típusként kezelik. Ha nem megfelelő típusú paramétert kapnak, azt az automatikus típuskonverzió át fogja alakítani. Feltüntettük emellett a függvények visszatérési értékének típusát is, bár azok általában többféle típusúak is lehetnek. Jelen függvények többsége sikeres adatbázis-művelet

esetén speciális, sikertelen művelet esetén logikai típusú (FALSE) értékkel tér vissza. A szintaxisokban [] jelek között levő paraméterek opcionálisak, elhagyhatóak. A függvények részletesebb leírása, példákkal együtt, a webes dokumentációban olvasható.

oci_connect, oci_pconnect

Szintaxis: `resource oci_connect(string $username, string $password [, string $db])`
`resource oci_pconnect(string $username, string $password [, string $db])`
Funkció: csatlakozás az adatbázishoz
Visszatérés: a kapcsolat leírója, hiba esetén FALSE.

A két függvény mindössze egyetlen részletben tér el egymástól: oci_connect mindig új kapcsolatot nyit a webkiszolgálótól az adatbázis felé, amit oci_close-zal le kell zárni, míg az oci_pconnect csak akkor nyit kapcsolatot, ha előzőleg ugyanazon paraméterekkel még nem nyitottak ugyanolyan típusút. Az efféle leírót PHP-ból nem lehet lezárni, az oldal futása után továbbra is nyitva marad, emiatt használata erős igénybevételnek kitett rendszerek esetén ellenjavallt.

oci_close

Szintaxis: `bool oci_close(resource $connection)`
Funkció: adatbázis-csatlakozás bontása

oci_parse

Szintaxis: `resource oci_parse(resource $conn, string $query)`
Funkció: SQL utasítás előkészítése végrehajtásra
Visszatérés: a végrehajtható utasítás leírója, hiba esetén FALSE

oci_bind_by_name

Szintaxis: `bool oci_bind_by_name(resource $statement, string $ph_name, mixed &$variable[, int $length[, int $type]])`
Funkció: PHP változó kötése egy Oracle értékhez
Visszatérés: TRUE ill. FALSE attól függően, hogy sikerült-e a művelet

A variable paraméter a PHP változó, a ph_name a kötés helye az SQL utasításban. A length a megadott változó maximális hossza, ha értéke -1, az aktuális hossz lesz a maximális hossz.

oci_execute

Szintaxis: `bool oci_execute(resource $statement[, int $mode]);`
Funkció: előkészített utasítás végrehajtása
Visszatérés: TRUE ill. FALSE attól függően, hogy sikerült-e a művelet

Ha a mód az alapértelmezett OCI_COMMIT_ON_SUCCESS, akkor az utasítás siker esetén automatikusan committál is. Ha az Oracle esetében megszokott tranzakciós viselkedést szeretnénk használni, *explicite* meg kell adnunk az OCI_DEFAULT módot. Ebben az esetben a tranzakció sorsát az oci_commit és oci_rollback utasításokkal szabályozhatjuk.

oci_free_statement

Szintaxis: `bool oci_free_statement(resource $stmt)`
Funkció: előkészített utasítással kapcsolatos erőforrások felszabadítása
Visszatérés: TRUE ill. FALSE attól függően, hogy sikerült-e a művelet

oci_fetch_array

Szintaxis: `array oci_fetch_array(resource $stmt[, int $mode])`
Funkció: a végrehajtott (`SELECT`) utasítás következő sorát olvassa be
Visszatérés: sikertelen művelet esetén `FALSE`, egyébként az eredményhalmaznak megfelelő tömbkifejezés

Ha a mód `OCI_ASSOC`, akkor a visszatérési érték egy asszociatív tömb lesz, melynek kulcsai az oszlopnevek, ha pedig a mód `OCI_NUM`, akkor egy (nullával kezdődő) egész számokkal indexelt tömb. Az alapértelmezett érték `OCI_BOTH` azaz a visszatérési értékül szolgáló tömbben az adatok mind az oszlopnevek, mint oszlopindexek segítségével elérhetőek. Az előző konstansok bármelyikével kombinálható (`+` jel segítségével) az `OCI_RETURN_NULLS`, aminek hatására üres értékeket is visszakapunk, azaz az üres cellák sem fognak hiányozni a tömbből, hanem a NULL értéket veszik fel. (Az ezzel való összehasonlítás az `===` operátorral vagy az `is_null` függvénnyel történhet.)

oci_num_fields

Szintaxis: `int oci_num_fields(resource $stmt)`
Funkció: a végrehajtott (`SELECT`) utasítás oszlopainak számát kérdezi le
Visszatérés: az oszlopok száma

oci_field_name

Szintaxis: `string oci_field_name(resource $stmt, int $col)`
Funkció: a végrehajtott (`SELECT`) utasítás megadott sorszámú oszlopának nevét kérdezi le. A sorszám egytől indul
Visszatérés: az oszlop neve.

oci_fetch_all

Szintaxis: `int oci_fetch_all(resource $stmt, array &$variable)`
Funkció: a végrehajtott (`SELECT`) utasítás összes eredménysorát kérdezi le egy kétdimenziós tömbbe. Az első index mindig a sor nullától indított sorszáma lesz, a második az oszlop (csupa nagybetűs) neve.
Visszatérés: sikertelen művelet esetén `FALSE`, egyébként a kiválasztott sorok száma

Mivel az automatikus típuskonverzió miatt a `0` (nulla) összetéveszthető a `FALSE` értékkel, és mindkettő érvényes visszatérési érték e függvény esetében, érdemes használni a típusazonosságot megkövetelő egyenlőség operátorát (`===`).

oci_error

Szintaxis: `array oci_error(resource $stmt|$conn)`
Funkció: a legutóbbi hiba jellemzőit kérdezi le
Visszatérés: a hibajellemzők tömbje

Ha `oci_parse` hibájának okára vagyunk kíváncsiak, az adatbázis-kapcsolat leíróját kell a paraméterbe tenni, míg ha `oci_execute` közben lépett fel hiba, az előkészített utasítás leírója adandó át.

5. Példaalkalmazás

Az alábbi fejezetrészekben egy elképzelt szállítmányozási cég belső, megrendelések felvételére és követésére szolgáló rendszere weboldalainak forráskódját mutatjuk be. A példa során egyrészt egy járműlistát jelenítünk meg (*vehicles.php*), illetve egy jármű részletes adatait kérdezzük le (*vehicle-details.php*). A példa nem teljes: a cél az, hogy illusztráljuk az

adatbázisból lekérdezett adatok megjelenítésének módját akár XHTML űrlapok segítségével bevitt paraméterek alapján.
Figyeljük meg, hogy a példaalkalmazásban – a mérés során is követendő módon – különválik az adatelérés és az adatok kimeneten való megjelenítése. Az adatelérés és az adatbázisból származó adatok megfelelő változókba történő betöltése PHP-val történik. A szerkesztett megjelenítés valamely weblapon a Smarty sablonkezelő rendszerrel valósul meg. A webes hivatkozáson keresztül elérhető *public_html* könyvtárstruktúrán belül csak a weboldalak generálásához minimálisan szükséges elemeket hagytuk, így ha egy kiszolgálóhiba miatt a PHP értelmező nem kerül meghívásra, ezáltal pedig egy támadó megtekintheti a kódot, olyan elemek, mint az adatbázis-csatlakozáshoz szükséges adatok vagy a lekérdezések szövegei már nem lesznek számára elérhetőek.
A *public_html* könyvtárban egyedül az *index.php* fájl található. Ez a *module* paraméterben kapja meg, hogy milyen műveleti sort kell végrehajtania (pl. járművek listáját vagy egy adott jármű részletezőoldalát jelenítse meg), és ennek megfelelően hívja be a *php_include* könyvtárból a *vehicles.php* vagy a *vehicle_details.php* oldalt.
A *php_include* könyvtár tartalmazza az alapvető konfigurációs beállításokat (adatbázis szolgáltatásazonosítója, adatbázis felhasználónév, adatbázisjelszó, illetve a különböző Smarty könyvtárak) tartalmazó és a vezérlésért felelős kitüntetett Smarty objektum példányosítását elvégző *config.php*-t, valamint a *smarty* könyvtáron belül négy újabb könyvtárat. Ezek közül számunkra a *templates* könyvtár a legfontosabb, hiszen ide kell elhelyeznünk a weboldalak szerkezetét leíró *.tpl* fájlokat. A könyvtárstruktúra tehát a következőképpen néz ki:

```
php_include
   vehicles.php
   vehicle-details.php
   config.php
   smarty
      cache
      configs
      templates
         error.tpl
         footer.tpl
         header.tpl
         index.tpl
         vehicles.tpl
         vehicle-details.tpl
      templates_c
public_html
  index.php
```

Ezek után nézzük a példaalkalmazás egyes fájljainak (kivonatolt) tartalmát.

5.1. config.php

```
<?php
// Az útvonalváltozók.
$_SmartyPath = "/usr/local/lib/php/Smarty/";
$_DocumentPath = "/usr/orauser/gabor/php_include/";

// A Smarty vezérlőosztály betöltése és példányosítása.
require($_SmartyPath."Smarty.class.php");
$Smarty = new Smarty();

// A Smarty konfigurációja.
$Smarty->template_dir = $_DocumentPath."smarty/templates/";
$Smarty->compile_dir = $_DocumentPath."smarty/templates_c";
$Smarty->cache_dir = $_DocumentPath."smarty/cache";
```

```
$Smarty->config_dir = $_DocumentPath."smarty/configs";
$Smarty->caching = 0;  // a cache kikapcsolasa

// Az adatbázishoz tartozó felhasználói név és jelszó, valamint a sid.
$user = "gabor_nagy";
$pass = "***";
$sid = "szglab";

// A weboldal karakterkészlete.
$xhtml_charset = "utf-8";

// Az adatbáziskapcsolat karakterkészlete.
$db_charset = "utf8";?>
```

5.2. index.php

```
<?php

// A konfigurációt tartalmazó állomány betöltése.
require("../../php_include/config.php");

// A kimenet generálása során használt tartalomtípus beállítása
...

// Kapcsolódás az adatbázishoz a config.php-ban beállított paraméterekkel
$connection = oci_connect($user, $pass, $sid, $db_charset);

// A sikeresség ellenőrzése. Ha nem sikerült, akkor a hiba megjelenítése.
if ($connection === FALSE) {
  // kritikus hiba: nem lehet kapcsolódni az adatbázishoz
  $error = oci_error();
  $Smarty->assign('ErrorMessage',htmlentities($error['message']));
  $Smarty->display('error.tpl');
  exit;
}

// A paraméterek feldolgozása.
// Ha nincs paraméter, akkor az index oldal látható, egyébként
// a module paraméternek megfelelő.
if(!isset($_GET['module'])) {
  // Az oldal cimenek megadasa.
  $Smarty->assign('PageTitle','Lakemacher and Co. Shipping Intl.');
  // A nyitolap sablonjanak megjelenitese.
  $Smarty->display('index.tpl');
} elseif($_GET['module'] == "vehicles") {
  include("../../php_include/vehicles.php");
} elseif($_GET['module'] == "vehicle-details") {
  include("../../php_include/vehicle-details.php");
}

// Az adatbáziskapcsolat lezárása
oci_close($connection);

?>
```

5.3. vehicles.php

```
<?php

// Táblázatos formában jeleníti meg a járművek listáját.
```

```
// Egy POST módon küldött űrlap "{$name}[]" nevű vezérlőeleméhez tartozó
// értékek listáját adja, vagy üres tömböt, ha nincs ilyen néven elérhető
// adat. Olyan esetekben használható, ha a vezérlőelemben több érték is
// kiválasztható, mint pl. a select vezérlőelem esetén multiple
// attribútummal.
function get_array_value_of_parameter($name, $default = array()) {
  if (array_key_exists($name, $_POST) && is_array($_POST[$name])) {
    // van ilyen paraméter a címsorban
    return $_POST[$name];
  } else {
    return $default;
  }
}

$days_of_week = get_array_value_of_parameter('days_of_week');

// Járművek listáját adja egy tömb soraiként.
$values = array();
foreach ($days_of_week as $day) {
  if ($day > 0 && $day <= 7) {
    $values[] = $day;
  }
}

if (count($values) > 0) {
  $value_list = implode(', ', $values);
  $condition = "TO_CHAR(purchase, 'd') IN ({$value_list})";
} else {
  $condition = '1 = 1';
}

// A megfelelő adatokat lekérdező SQL utasítás.
$query = <<<SQL
SELECT
  numberplate,
  TO_CHAR(purchase, 'mm/dd/yyyy') AS purchase_date,
  TO_CHAR(maintenance, 'mm/dd/yyyy') AS maintenance_date
FROM vehicles
WHERE {$condition}
SQL;

$statement = oci_parse($connection, $query);
if ($statement === FALSE) {
  $queryhtml = htmlentities($query);
  $errormsg = "<h1>Illegal query:</h1><p>{$queryhtml}</p>";
  $Smarty->assign('ErrorMessage',$errormsg);
  $Smarty->display('error.tpl');
  exit;
}

// Lehívja az eredményhalmaz meghatározott sorait.
$vehicles = array();
if (oci_execute($statement, OCI_DEFAULT)) {
  oci_fetch_all($statement, $vehicles, $skip, $maxrows,
    OCI_FETCHSTATEMENT_BY_ROW + OCI_ASSOC);
}

global $Smarty;

// A változók átadása a sablon használatához.
```

```
$Smarty->assign('PageTitle',"List of vehicles");
$Smarty->assign('days_of_weeks',$days_of_week);
$Smarty->assign('vehicles',$vehicles);

// Az oldal megjelenítése.
$Smarty->display('vehicles.tpl');

?>
```

5.4. vehicle_details.php

```
<?php

// Táblázatos formában jeleníti meg egy jármű részletes adatait.

// Egy címsorban átadott egész paraméter numerikus értékét adja vissza.
function get_integer_value_of_parameter($name, $default = FALSE) {
  if (array_key_exists($name, $_GET)) {  // van ilyen paraméter a címben
    $value = $_GET[$name];
    if (is_numeric($value)) {  // ami számérték
      return (int) $value;  // csonkolás egész értékre
    }
  }
  return $default;
}

// Egy címsorban átadott szöveges paraméter értékét adja vissza.
function get_string_value_of_parameter($name, $default = FALSE) {
  // globális változók függvénytörzsön belüli használata a global
  // direktívával lehetséges,a $_GET, $_POST és $_REQUEST esetén azonban
  // erre nincs szükség
  if (array_key_exists($name, $_GET)) {  // van ilyen paraméter a címben
    return $_GET[$name];  // a $_GET tömb füzéreket tartalmaz
  } else {
    return $default;
  }
}

// Egy kódhoz tartozó szöveges elnevezést ad vissza.
function get_type_string($type) {
  switch ($type) {
    case 'g': return 'gasoline tank';
    case 'c': return 'container chassis';
    case 'd': return 'dump';
    case 'v': return 'dry freight van';
    case 'l': return 'livestock trailer';
    case 'a': return 'auto carrier';
    default: return 'not specified';
  }
}

$numberplate = get_string_value_of_parameter('numberplate');

// Az adatok lekérdezése az adatbázisból.
if ($numberplate !== FALSE) {

  $query =
    "SELECT ".
      "numberplate, ".
      "vehicle_type, ".
      "TO_CHAR(purchase, 'mm/dd/yyyy') AS purchase_date, ".
```

```
      "TO_CHAR(maintenance, 'mm/dd/yyyy') AS maintenance_date ".
    "FROM vehicles ".
    "WHERE numberplate = :numberplate";

  $statement = oci_parse($connection, $query);
  if ($statement === FALSE) {
    $queryhtml = htmlentities($query);
    $errormsg = "<h1>Illegal query:</h1><p>{$queryhtml}</p>";
    $Smarty->assign('ErrorMessage',$errormsg);
    $Smarty->display('error.tpl');
    exit;
  }

  // paraméterérték beállítása kötéssel
  oci_bind_by_name($statement, ':numberplate', $numberplate);

  $details = array();
  if (oci_execute($statement, OCI_DEFAULT)) {
    oci_fetch_all($statement, $details, $skip, $maxrows,
      OCI_FETCHSTATEMENT_BY_ROW + OCI_ASSOC);
  } else {
    $details = FALSE;
  }
  if (count(details) > 0) {
    $details = $details[0]; // ennek az eredményhalmaznak max. egy sora van
    $vehicle = 1;
  } else {
    $details = FALSE;
    $vehicle = 0;
  }
} else {
  $details = FALSE;
}

// A változók átadása.
$Smarty->assign('PageTitle','details');
$Smarty->assign('vehicle',$vehicle);
$Smarty->assign('numberplate',$details['NUMBERPLATE']);
$Smarty->assign('type',get_type_string($details['VEHICLE_TYPE']));
$Smarty->assign('purchase_date',$details['PURCHASE_DATE']);
$maintenance_date = $details['MAINTENANCE_DATE'];
$maintenance_date = $maintenance_date === NULL ?
  "&lt; field not specified &gt;" : $maintenance_date;
$Smarty->assign('maintenance_date',$maintenance_date);

// A felület megjelenítése.
$Smarty->display('vehicle-details.tpl');
?>
```

5.5. index.tpl

```
{*
 * A főoldal Smarty-val megvalósítva.
 *}

{php}
  $this->assign('CssFiles', Array("style.css"));
  $this->assign('JSFiles', Array());
{/php}

{include file='header.tpl'}
```

```
{include file='menu.tpl'}
<h1>Welcome to the web site of
  Lakemacher &amp;amp; Co. Shipping International!</h1>

{include file='footer.tpl'}
```

5.6. vehicle-details.tpl

```
{*
 * A Vehicles oldal Smarty-val megvalósítva.
 *}

{php}
// A sablonfájlhoz tartozó stílus- és JavaScript fájlok megadása.
  $this->assign('CssFiles', Array("style.css"));
  $this->assign('JSFiles', Array());
{/php}

// Fejléc előállítása.
{include file='header.tpl'}
{include file='menu.tpl'}

{*
 * Ha van találat, akkor az átadott értékek megjelenítése.
 *}
{if $vehicle neq 0}
  <table>

  <tr>
  <td>Numberplate:</td>
  <td>{$numberplate}</td>
  </tr>

  <tr>
  <td>Type of vehicle:</td>
  <td>{$type}</td>
  </tr>

  <tr>
  <td>Date of purchase:</td>
  <td>{$purchase_date}</td>
  </tr>

  <tr>
  <td>Date of last maintenance check:</td>
  <td>{$maintenance_date}</td>
  </tr>

  </table>
{else}
  <p>No such vehicle, invalid identifier.</p>
{/if}

{include file='footer.tpl'}
```

5.7. vehicles.tpl

```
{*
 * A járműveket listázó oldal Smarty-val megvalósítva.
 *}
```

```
// A sablonfájlhoz tartozó stílus- és JavaScript fájlok megadása.
{php}
  $this->assign('CssFiles', Array("style.css"));
  $this->assign('JSFiles', Array());
{/php}

// Fejléc előállítása.
{include file='header.tpl'}
{include file='menu.tpl'}
  <form method="post">
    <table>

    <tr>
    <td>Show vehicles purchased on</td>

    <td>
    <select name="days_of_week[]" multiple="multiple">
    {*
     * Az űrlapon kiválasztott értékek alapján a napok kijelölése.
     *}
    <option {if array_search(1, $days_of_weeks) !== FALSE}
        selected="selected" {/if} value="1">Monday</option>
    <option {if array_search(2, $days_of_weeks) !== FALSE}
        selected="selected" {/if}value="2">Tuesday</option>
    <option {if array_search(3, $days_of_weeks) !== FALSE}
        selected="selected" {/if}value="3">Wednesday</option>
    <option {if array_search(4, $days_of_weeks) !== FALSE}
         selected="selected" {/if}value="4">Thursday</option>
    <option {if array_search(5, $days_of_weeks) !== FALSE}
        selected="selected" {/if}value="5">Friday</option>
    <option {if array_search(6, $days_of_weeks) !== FALSE}
        selected="selected" {/if}value="6">Saturday</option>
    <option {if array_search(7, $days_of_weeks) !== FALSE}
        selected="selected" {/if}value="7">Sunday</option>
    </select>
    </td>
    </tr>

    <tr>
    <td colspan="2">
    <input type="submit" />
    </td>
    </tr>

    </table>

  </form>

    <table>

      <tr>
      <th>Numberplate</th>
      <th>Date of purchase</th>
      <th>Date of last maintenance check</th>
      </tr>

      {*
       * Az adatbázisból kapott, és tömbben átadott
       * értékek megjelenítése egy ciklussal.
       *}
```

```
      {foreach item=record from=$vehicles}
      <tr>
      <td><a href="index.php?module=vehicle-details&amp;
         numberplate={$record.NUMBERPLATE|escape:'url'}">
         {$record.NUMBERPLATE}</a></td>
      <td>{$record.PURCHASE_DATE}</td>
      <td>{$record.MAINTENANCE_DATE}</td>
      </tr>

      {foreachelse}
      Nincs találat.

      {/foreach}

    </table>

{include file='footer.tpl'}
```

5.8. header.tpl

```
{*
 * Az XHTML oldal fejléce. A tömbökben átadott CSS és JavaScript fájlok
 * betöltése, valamint az oldal fejlécének behelyettesítése
*}

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>{$PageTitle}</title>

{section name=seq loop=$CssFiles}
<link rel="stylesheet" type="text/css" href="{$CssFiles[seq]}" />
{/section}

{section name=seq loop=$JSFiles}
<script type="text/javascript"
src="javascript/{$JSFiles[seq]}.js"></script>
{/section}
</head>

<body>
```

5.9. footer.tpl

```
{*
 * Az XHTML oldal lezárása.
 *}
<hr/>
<p>Copyright &copy; 2009 Lakemacher &amp; Co. Shipping International</p>
</body>
</html>
```

5.10. error.tpl

```
{*
 * A hibamegjelenítő sablon.
 *}
```

```
{php}
  $this->assign('CssFiles', Array("style.css"));
  $this->assign('JSFiles', Array());
{/php}
{include file='header.tpl'}
{include file='menu.tpl'}
<pre style="border: 1px dotted black; background-color: #CCC;">
{$ErrorMessage}
</pre>
{include file='footer.tpl'}
```

5.11. menu.tpl

```
{*
 * A menu megjelenitese.
 *}
<p>
Page:
<a href="index.php?module=vehicles">vehicles</a> |
Source:
<a href="source-code.php?file=index.php">index.php</a> |
<a href="source-code.php?file=vehicles.php">vehicles.php</a> |
<a   href="source-code.php?file=vehicle-details.php">vehicle-details.php</a>
|
Templates:
<a href="source-code.php?file=header.tpl">header.tpl</a> |
<a href="source-code.php?file=footer.tpl">footer.tpl</a> |
<a href="source-code.php?file=error.tpl">error.tpl</a> |
<a href="source-code.php?file=index.tpl">index.tpl</a> |
<a href="source-code.php?file=menu.tpl">menu.tpl</a> |
<a href="source-code.php?file=vehicles.tpl">vehicles.tpl</a> |
<a   href="source-code.php?file=vehicle-details.tpl">vehicle-details.tpl</a>
|
Initial lab setup:
<a href="source-code.php?file=skeleton-index.php">index.php</a> |
<a href="source-code.php?file=skeleton-index.tpl">index.tpl</a> |
<a href="source-code.php?file=skeleton-config.php">config.php</a>
</p>
```

I. Függelék: UNIX összefoglaló a legfontosabb parancsokról

1. Könyvtárstruktúra

- `mkdir <dirname>` létrehoz egy könyvtárat.
- `rmdir <dirname>` törli a könyvtárat.
- `cd <dirname>` belép a könyvtárba.
- `ls <dirname>` kilistázza a könyvtár tartalmát. Az `ls -l` paranccsal részletes listát kapunk. Néhány speciális könyvtár:
  - `.` aktuális könyvtár,
  - `..` szülő könyvtár,
  - `/` gyökérkönyvtár,
  - `~` a `$(HOME)` könyvtár – a felhasználó munkakönyvtára.

  Több `cd` parancs egymás utáni kiadása helyett `cd <dirname>/<dirname>/[stb]` alakú parancs is kiadható.
- `cp <source> <dest>` fájlt (fájlokat) másol.
- `rm <filename>` fájlt töröl.
- `mv <source> <dest>` fájlt mozgat.
- `cat <filename>` fájl tartalmának megjelenítése.

2. Jogosultságok

UNIXban a fájlokra három szinten háromfajta jogosultságot lehet beállítani. A szintek:

- owner – az állomány gazdája (`u`)
- group – az állomány csoportja (`g`)
- other – a csoporton kívül mindenki más (`o`)

A fajták (könyvtárra vonatkozó jog zárójelben):

- **`r`**`ead` – olvasási (listázási) jog
- **`w`**`rite` – írási jog
- e**`x`**`ecute` – futtatási (belépési) jog

A gazda és csoport a `chown <owner>.<group> <filename>` paranccsal állítható, A jogosultságok állítására a `chmod <szint(ek)>(+|-)<fajta('k)> <filename>` parancs szolgál. Például a foo.php fájlra olvasási jog adása mindenki más számára: `chmod o+r foo.php`

3. Fájlkezelő

- `mc` Midnight Commander. Hasonló, mint az nc/FAR DOS/Windows alatt. Támogatja többek között az ftp átvitelt, könyvtárak ki-/betömörítését.

4. Szövegszerkesztők

- `pico` szövegeket jó benne szerkeszteni, forrást nem annyira, mert automatikusan sort tör.
- `joe` jó forrás szerkesztésre, beállítható, hogy ne törjön sort, hozzá kell szokni a ^K+H típusú parancsokhoz, vagyis [CTRL]+[K] után egy [H] megnyomása (jelen esetben az a help parancs).
- `mcedit` az mc szövegszerkesztője. Ncurses alapon syntax highlighthingot, azaz kódszínezést biztosít.
- `vi` ősi unixos szövegszerkesztő. Kezdőknek nem ajánlott.

5. Grafikus szövegszerkesztők

- `emacs` sokan dicsérik, mert sokat tud.
- `nedit` kevesebbet tud, de kezdőknek egyszerűbb megtanulni.

6. Egyebek

- `gzip, gunzip` tömörítő/kitömörítő program. Csak egy fájlt kezel, ezért ha több fájlt kell egybe zippelni, akkor azokat előtte egybe kell csomagolni.
- `tar` csomagolóprogram. Becsomagolás: `-cvf <tarfile> <source>`, kicsomagolás: `-xvf <tarfile> <dest>`.
- csomagolás és tömörítés egyben: `tar -zcvf <tgzfile> <sources>`, visszafelé: `tar -zxvf <tgzfile> <dest>`
- `ssh` biztonságos távoli shell. Pl: `ssh -X xy123@ural2.hszk.bme.hu`
- `scp <source> <dest>` biztonságos fájlátvitel.
  Pl: `scp xy123@rapid.eik.bme.hu:php.tgz ./`
- `ping <host>` hostokat teszteli, hogy elérhetőek-e.

A parancsokról bővebb információ a `man <command>` begépelésével tudható meg.

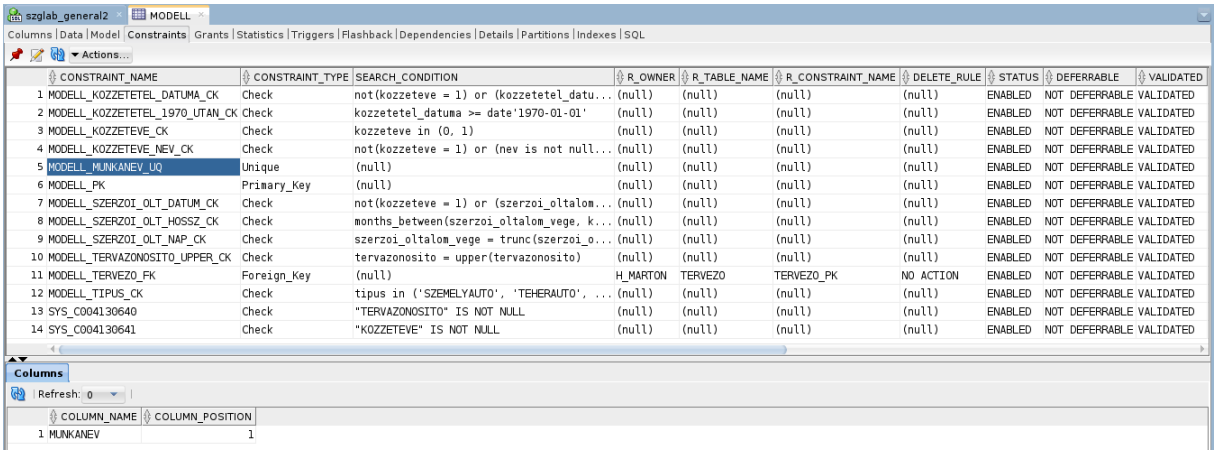
II. Függelék: Adatbázis kényszerek az Oracle-ben

Kiegészítés az I. labor anyagához

A constraintek, vagyis kényszerek egyszerű előírások, szabályok az adatbázisban található adatokra nézve, melyek elősegítik az ellentmondásmentesség (általában a tágabb értelemben vett "konzisztencia") fenntartását az adatbázis-szerver szintjén. Sőt, a kényszerek ismeretében az Oracle képes a lekérdezéseket is jobban optimalizálni. Ilyen szabály lehet például egy mező (attribútum) adatainak szintaktikai ellenőrzése (pl. személyi igazolvány szám: két betű, hat szám formátumú (régi, füzet formájú) vagy fordított sorrendben (új, kártya formájú) legyen), de több adat összefüggését is ellenőrizhetjük (az adatok közötti hivatkozások helyességének fenntartása érdekében). A kényszerek tehát igen hasznosak tudnak lenni, ugyanakkor – tapasztalat szerint – egy rendszer módosítása során sok gondot is tudnak okozni.

Egy táblához több kényszer is megadható. Definiálásukra alapvetően két lehetőségünk van: vagy a megfelelő SQL parancsot gépeljük be vagy valamilyen grafikus eszközt használunk a parancs kényelmes összeállítására, így nem kell ismernünk a pontos szintaxist. Az SQL Developerben új tábla „Advanced" módú felvételekor, vagy meglévő tábla módosításakor a Constraints fül alatt találhatóak a kényszerek, és az Actions/Constraints parancsokkal módosíthatók/törölhetők vagy adhatók a táblához újabbak.

Tulajdonképpen az is kényszer, ha egy mező értéke nem lehet NULL, vagyis ha a tábla definíciójában szerepel a NOT NULL kulcsszó. Ezt a tábla szerkesztésénél a mezők megadása során állíthatjuk be, és a constraintek sorában is feltűnik (l. az 1. ábrán).



1. ábra: Táblán definiált kényszerek megjelenítése az SQL Developerben
A teljes példa a függelék végén szerepel

A képernyő alapvetően két részre tagolható. A felső táblázatban jelennek meg a kényszerek alapadatai (név, típus, feltétel, a hivatkozás részletei stb.), az alsó táblázatban pedig az, hogy egy-egy kényszer mely oszlopokra vonatkozik. Az alsó táblázat tartalma mindig a felső táblázat egy-egy sorára vonatkozik.

A felső táblázatban az alábbi fontosabb oszlopok szerepelnek:
Constraint Name – a kényszer neve, amit létrehozáskor megadtunk. Amennyiben nem adtunk meg nevet, a rendszer automatikusan generál nagyjából véletlenszerű karakterláncot (pl. az ábrán látható „not null” constraint neve is így keletkezett). Noha beszédes név megadása nem kötelező mégis célszerű azt használni: ha ugyanis valamilyen későbbi művelet azért nem végrehajtható, mert sértené a kényszert, akkor a rendszer ezzel a névvel fog hivatkozni a kényszerre. Kellően beszédes név esetén tehát azonnal tudni fogjuk, mi a probléma, anélkül hogy hosszasan keresgélnünk kellene. Érdemes elnevezési konvenciót is használni, pl. a kényszer típusától függő végződéssel ellátni a neveket (pl. primary key: _PK, unique: _UQ, check: _CK, foreign key: _FK)

Constraint Type – A kényszer típusa, mely Oracle-ben az alábbi lehet: UNIQUE, PRIMARY_KEY, FOREIGN_KEY, CHECK. A kényszer típusa alapvetően megszabja, hogy milyen paramétereket kell megadni létrehozásukkor.
Search Condition – CHECK típusú kényszernél használatos. Lásd alább.
R Owner – FOREIGN típusú kényszernél használatos. Lásd alább.
R Table Name – FOREIGN típusú kényszernél használatos. Lásd alább.
R Constraint Name – FOREIGN típusú kényszernél használatos. Lásd alább.
Delete Rule – FOREIGN típusú kényszernél használatos. Lásd alább.

Status – Enabled v. Disabled értéket vehet fel. Ha Disabled, akkor ideiglenesen letiltjuk, vagyis a rendszer úgy tekinti, mintha ez a kényszer egyáltalán nem is létezne. Célszerű például ideiglenesen kikapcsolni egy kényszert, ha nagyon sok, a kényszert nem sértő adatot importálunk egyszerre – így gyorsabb lesz a feldolgozás.

Deferrable – *Deferrable* v. *Not Deferrable* értéket vehet fel. Amennyiben Deferrable értékre állítjuk, úgy a felhasználó (vagy a kapcsolódó alkalmazás) kérheti a kényszer késleltetését (l. `set constraint` parancs). Ennek eredménye, hogy a kényszer megsértése esetén a rendszer nem a kényszert sértő utasítás után azonnal ad hibajelzést, hanem csak a tranzakció végén, vagyis a COMMIT utasítás kiadásakor. Ennek nagy előnye, hogy a tranzakció elemi lépései után még sérülhet ugyan a kényszer (pl. egy rekord módosítása több lépésben, mezőnként), de természetesen a tranzakció végén már nem.

(Initially) Deferred – *Deferred* v. *Immediate* értéket vehet fel. Bár nem jelenik meg ebben a táblázatban, a kényszerek egy olyan fontos tulajdonságát írja le, mely csak akkor értelmezett, ha a Deferrable mezőt *Deferrable* értékűre állítottuk. Amennyiben ez a mező is igaz (*deferred*) értékű, akkor a rendszer automatikusan késlelteti a kényszer ellenőrzését a tranzakció végéig, vagyis a felhasználónak nem kell külön kérnie ezt a viselkedést.

Validated – *Validated* v. *Not Validated* értéket vehet fel. Amennyiben értéke *Not Validated*, akkor a rendszer a kényszer *Status=Enabled* állapotában csak az új és módosított rekordokat ellenőrzi. Míg ha értéke *Validated*, a rendszer garantálja, hogy a meglévő rekordok is eleget tesznek a kényszernek.

1. CHECK kényszer

CHECK típusú kényszer esetén a Check Condition mezőt kell kitölteni. Ide egy logikai kifejezést adhatunk meg, hasonlóan mint a lekérdezések WHERE clause-ában. A rendszer rekordok felvételénél és módosításánál ellenőrzi, hogy ez a feltétel teljesül-e, és ha nem, azaz

a kényszer sérül, a végrehajtást a rendszer megtagadja. A kifejezésben sajnos nem használhatunk subquery-ket, vagyis bonyolultabb lekérdezéseket, melyek más táblákra is hivatkozhatnának. A feltétel tehát csakis a kényszerhez tartozó tábla egy vagy néhány mezőjét vizsgálhatja. Nem szerepelhetnek továbbá nem determinisztikus visszatérési értékű függvények hívásai (pl. `sysdate`) sem a feltételben.
Megjegyzés: az Oracle Database terminológiája szerint a CHECK kényszer teljesül, ha az abban szereplő kifejezés kiértékelés eredménye igaz vagy ismeretlen (UNKNOWN), ami tipikusan NULL értékű mezők miatt fordul elő.

A kényszerekben szerepeltethető kifejezésre néhány példát kiemeltünk a a függelék végén közölt demonstrációból.

Rögzített értékkészletű mező („enum").
```
tipus in ('SZEMELYAUTO', 'TEHERAUTO', 'VIZI_JARMU', 'LEGI_JARMU')
```
A fenti megjegyzés értelmében a **`tipus`** mező a felsorolt 4 értéket veheti fel, vagy üres lehet (NULL értékű).
Rögzített értékkészletű mező, ami nem lehet NULL.
Ezt két lépésben érjük el:

a meződefinícióban:	`kozzeteve number(1) default 0 not null`
a feltétel	`kozzeteve in (0, 1)`

A mezőben szereplő betűk nagybetűk legyenek.
```
tervazonosito = upper(tervazonosito)
```
A dátum 1970. január 1 vagy későbbi.
```
kozzetetel_datuma >= date'1970-01-01'
```
A dátum nap pontosságú (az idő komponense 00:00:00).
```
szerzoi_oltalom_vege = trunc(szerzoi_oltalom_vege)
```
Az első dátum max 80 évvel későbbi a másodiknál.
```
months_between(szerzoi_oltalom_vege, kozzetetel_datuma) <= 80*12
```
Ha a kozzeteve mezőben 1 szerepel, akkor a kozzetetel_datuma nem üres.
A „ha X akkor Y" implikáció ekvivalens a „nem(X) vagy Y" alakkal, ezért:.
```
not(kozzeteve = 1) or (kozzetetel_datuma is not null)
```

2. UNIQUE kényszer

A UNIQUE típusú kényszerrel tulajdonképpen egy kulcsot definiálunk a táblához. Hatása kettős: a kulcsban szereplő mezők értekei által képzett kombinációnak egyedinek kell lennie a táblában. A rendszer alapértelmezésben automatikusan létrehoz egy indexet a megadott mezők alapján az egyediség ellenőrzésére és a keresések gyorsítására. A kulcshoz tartozó mező(ke)t az alsó táblázat Table Colums oszlopában találjuk.
A UNIQUE kényszerben szereplő mezők egyéb tiltás hiányában felvehetnek NULL értéket. Ilyen a függelék végén közölt pélában szereplő modell tábla munkanev attribútuma is.

Megjegyzés: egy táblában több olyan rekord is szerepelhet, amely a UNIQUE kényszerben szereplő összes mezőjében a NULL értéket tartalmazza.

3. PRIMARY kényszer

A PRIMARY típusú kényszer az elsődleges kulcsot jelöli. Hasonló a UNIQUE típusú kényszer, de itt valamivel több megkötéssel találkozunk. Egyrészt egy táblára több UNIQUE, de csak egyetlen PRIMARY kényszer létezhet. Másrészt, míg előbbi megengedi a NULL értékeket, a PRIMARY kényszer egyúttal azt is biztosítja, hogy a megadott mezőkben ne

lehessen NULL érték. Harmadrészt pedig általában az elsődleges kulcs azonosítja egyértelműen a rekordot a táblában, ezért idegen kulcsokkal erre hivatkozunk más táblából. Létrehozása ugyanúgy történik mint a UNIQUE kényszernél, és ez is indexet hoz létre a háttérben.
Példa: a függelék végén közölt **`modell`** tábla **`modell_id`** attribútumán **`modell_pk`** néven definiáltunk egy elsődleges kulcs kényszert.

4. FOREIGN kényszer

A FOREIGN típusú kényszer a legérdekesebb és legbonyolultabb kényszer az Oracle repertoárjában. Idegen kulcsot jelöl, tehát az adott táblában bizonyos mezők – egy másik tábla bizonyos mezői alapján – hivatkoznak a másik tábla rekordjaira. A hivatkozott táblát általában szülő, a hivatkozó táblát gyermek táblának szokták hívni. Az idegen kulcshoz tartozó kényszert a gyermek táblában definiáljuk.
Idegen kulcs megadása esetében a Referenced Schema és Referenced Table mezőkben kell megadni, hogy mely séma mely táblájának rekordjaira fogunk hivatkozni az éppen szerkesztett táblából, vagyis itt adjuk meg a szülő táblát (az SQL Developerben ezt érdemes a tábla létrehozásakor azonnal megtenni). A szülő tábla egy Primary vagy Unique kényszerét azonosítva segít az SQL Developer abban, hogy hány oszlopból álló idegen kulcsot kell létrehozni a gyermek táblában.
Ezután az alsó táblázatban adhatjuk meg a hivatkozó és hivatkozott mező(k) páros(ai)t. A gyermek táblában szereplő hivatkozó mezőt a Local Columns, a szülő táblában szereplő hivatkozottat a Referenced Columns oszlopban.
Amennyiben a kényszerhez a Cascade On Delete paramétert bejelöljük, akkor törlési láncot is létrehozunk. Ilyenkor ha egy rekordot kitörlünk a szülő táblában, a rendszer automatikusan törli a gyermek táblában azon rekordokat, melyek az eredetileg törölni kívánt rekordokra hivatkoztak. Másik lehetőség a „set to null” amelynek hatására a szülő táblában történő törlés esetén a gyermek tábla megfelelő bejegyzései null értékre kerülnek beállításra. A „restrict” (vagy a fenti ábra táblázatában: „no action”) alapbeállítás pedig nem enged szülőbejegyzést törölni, amíg van rá hivatkozó rekord a gyermek táblában.
Példa: a függelék végén közölt példában a **`modell`** tábla **`tervezo`** attribútuma a tervezo tábla **`tervezo_id`** mezőjére mutat. Ezt írja le a **`modell.modell_tervezo_fk`** kényszer.

5. Teljes példa SQL nyelven

Az alábbiakban jármű-modelleket és tervezőit leíró táblapár SQL-nyelvű definícióját közöljük a különböző kényszertípusok demonstrálására.

```
create table tervezo (
  tervezo_id number not null
, nev nvarchar2(200) not null
, szuletett date
, constraint tervezo_pk primary key (tervezo_id)
);
comment on table tervezo is 'A tervezők alapadatait tároló tábla.';
```

```
create table modell (
  modell_id number
, tervezo number
, tervazonosito nvarchar2(30) not null
, munkanev nvarchar2(200)
, tipus nvarchar2(30)
, kozzeteve number(1) default 0 not null
, nev nvarchar2(200)
, kozzetetel_datuma date
, szerzoi_oltalom_vege date
, constraint modell_pk primary key (modell_id)
, constraint modell_tervezo_fk foreign key (tervezo)
    references tervezo (tervezo_id)
, constraint modell_tervazonosito_upper_ck check
    (tervazonosito = upper(tervazonosito))
, constraint modell_munkanev_uq unique (munkanev)
, constraint modell_tipus_ck check
    (tipus in ('SZEMELYAUTO', 'TEHERAUTO', 'VIZI_JARMU', 'LEGI_JARMU'))
, constraint modell_kozzeteve_ck check (kozzeteve in (0, 1))
-- ha közzéttett a modell, akkor nem üres:
--   nev, kozzetetel_datuma, szerzoi_oltalom_vege
, constraint modell_kozzeteve_nev_ck check
    (not(kozzeteve = 1) or (nev is not null and length(nev)>0))
, constraint modell_kozzetetel_datuma_ck check
    (not(kozzeteve = 1) or (kozzetetel_datuma is not null))
, constraint modell_szerzoi_olt_datum_ck check
    (not(kozzeteve = 1) or (szerzoi_oltalom_vege is not null))
, constraint modell_kozzetetel_1970_utan_ck check
    (kozzetetel_datuma >= date'1970-01-01')
, constraint modell_szerzoi_olt_nap_ck check
    (szerzoi_oltalom_vege = trunc(szerzoi_oltalom_vege))
, constraint modell_szerzoi_olt_hossz_ck check
    (months_between(szerzoi_oltalom_vege, kozzetetel_datuma) <= 80*12)
);
comment on table modell is 'Közúti, vízi és légijárművek adatait tároló
tábla.';
comment on column modell.tervazonosito is 'Nagybetűkkel írandó.';
comment on column modell.munkanev is 'A modell egyedi munkaneve (belső
neve). Lehet üres (null), ha még nem választott a tervező.';
comment on column modell.tipus is 'Lehetséges típusok személyautó
(SZEMELYAUTO), tehergépjármű (TEHERAUTO), vízi jármű (VIZI_JARMU) ill. légi
jármű (LEGI_JARMU).';
comment on column modell.kozzeteve is 'A 0, 1 értékek rendre a Hamis, Igaz
logikai értéket kódolják.';
comment on column modell.nev is 'A modell neve. Ha közzétett, akkor nem
lehet üres, tehát sem null, sem a 0-hosszú üres sztring.';
comment on column modell.kozzetetel_datuma is 'A közzététel időpontja. Ha
közzétett, akkor nem lehet üres. Csak 1970-ben vagy később közzétett
modelleket tárolunk.';
comment on column modell.szerzoi_oltalom_vege is 'A szerzői oldatol
lejárati napja. Ha közzétett, akkor nem lehet üres. A közzététel dátumához
képest 80 éven belüli dátum. Nap pontosságú, és az oltalom a következő
napon szűnik meg.';
```

III. Függelék: Reguláris kifejezések

Szerző: Soproni Péter (sopronipeter@db.bme.hu)

1. Bevezetés

Szinte a számítástechnikával egyidős az igény egy olyan nyelv kialakítására, amely egyszerű szűréseket, átalakításokat, ellenőrzéseket tesz lehetővé alapvetően szöveges adatokon, deklaratív szemlélet mellett. Az idők során több nyelv és ezen nyelvek több nyelvjárása alakult ki a különböző szabványosítási kísérletek ellenére is [R1-R2].

Jelenleg a legelterjedtebb a Perl nyelv által bevezetett szintaxis [R2]. Ennek kisebb-nagyobb mértékben átdolgozott változatait szinte az összes modern programozási nyelv támogatja (.NET [R3], Java [R4], PHP [R5]). A továbbiakban mi is ennek a nyelvnek az alapjait ismertetjük, illetve a hozzá Java-ban biztosított API használatába nyújtunk betekintést.

2. Alapszabály

Reguláris kifejezések írása során a cél egy minta megfogalmazása. Az adott mintára illeszkedő vagy éppen nem illeszkedő szövegrészeket keressük. Minden a mintában szereplő karakter, ha nem speciális, egy vele azonos karakter létét írja elő az illeszkedésekben.

A legfontosabb speciális karakterek: *[]().\^+ ?*{},$-*. Ha ezekre, mint nem speciális karakterre kívánunk illeszkedést biztosítani, úgy eléjük visszaperjelet (backslash) írni, azaz például a '\.' illeszkedik a pontra, a '\\' pedig a visszaperjelre.

Ennek megfelelően, ha egy ismert keltezésére illeszkedő szakaszokat keresünk (azaz például egy olyan szövegrészt, ami egy ismert kibocsátási hellyel kezdődik, és egy dátummezővel záródik), azt a következő reguláris kifejezéssel tehetjük meg:

Reguláris kifejezés:[38] '[Pécs, 1978\. december 07\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Bécs, 1978. december 07.
	Pécs,1978. december 07.
	Pécs,1978, december 07.

Látható, hogy a pontra való illeszkedés érdekében, mivel az egy speciális karakter, egy visszaperjelet kellett elé írni. További fontos észrevétel, hogy itt a szóköznek is jelentése van. A többi karakterhez hasonlóan egy vele azonos írásjel meglétét követeli az illeszkedés fennállásához (ezért nem illeszkedik a második ellenpélda).

3. Karakterosztályok

A legtöbb esetben a keresett szövegrészben egy adott helyen nem egyetlen karakter, hanem karakterek egy halmaza állhat. A reguláris kifejezésekben ennek megfogalmazására létrehozott konstrukció az úgynevezett karakterosztály.
Általános szintaxis: *[<karakterosztály elemei>]*
Az előző példánál maradva, általában nem egy konkrét keltezést keresünk, hanem több lehetőségből egyet. Ha mind Pécset, mind Bécset el tudjuk fogadni, mint helyszínt, akkor kifejezésünk a következőképpen alakul:

Reguláris kifejezés: '[BP]écs, 1978\. december 07\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Budapest, 1978. december 07.
Bécs, 1978. december 07.	Mécs, 1978. december 07.

3.1. Egyszerű karakterosztály

A lehetséges karakterek egyszerű felsorolásával képezzük. Minden felsorolt karakterre illeszkedik, de semmilyen más elemre nem.
Szintaxis: *[<az elfogadott karakterek listája>]*

Reguláris kifejezés: '[PBb]'	
Illeszkedésre példa	**Nem illeszkedő példa**
b	p
P	PP

3.2. Kizárás

Lehetőség van arra, hogy azt fogalmazzuk meg, mit nem fogadunk el az adott helyen.
Szintaxis: *[^<el nem fogadott karakterek>]*

Reguláris kifejezés: '[^PBb]'	
Illeszkedésre példa	**Nem illeszkedő példa**
G	b
V	vv

3.3. Intervallum karakterosztály

Az egyes elfogadott karakterek helyett az elfogadott intervallumokat[39] is megadhatjuk. Ilyen intervallum lehet pl. a számok halmaza 1 és 7 között, vagy az összes kisbetű.
Szintaxis: *[<intervallum alsó széle>-<intervallum felső széle>]*

[38] A reguláris kifejezésekre mutatott példáknál a kifejezéseket határoló aposztrófok csak a kifejezés határait jelölik ki, nem képezik azok részét. A példáknál, ha külön nem jelezzük, a megadott szöveg teljes illeszkedését vizsgáljuk, nem azt, hogy van-e illeszkedő részminta vagy részszöveg.
[39] Az intervallum a két határoló karakter ASCII karakterkódja által meghatározott intervallumba eső karakterkóddal meghatározott karakterek halmaza.

Reguláris kifejezés: '[1-9]'	
Illeszkedésre példa	**Nem illeszkedő példa**
3	0
4	44

3.4. Karakterosztályok uniója

Az egyes karakterosztályok uniója alatt azokat a karaktereket értjük, amelyek valamely eredeti karakterosztályban benne vannak.
Szintaxis: *[<első karakterosztály>|<második karakterosztály>]* vagy *[<első karakterosztály><második karakterosztály>]*[40]

Reguláris kifejezés: '[1-9|ab]'	
Illeszkedésre példa	**Nem illeszkedő példa**
a	0
7	A

3.5. Karakterosztályok metszete

Az egyes karakterosztályok metszete azon karakterek halmaza amelyek mindkét karakter osztályban megtalálhatók.
Szintaxis: *[<első karakterosztály>&&<második karakterosztály>]*

Reguláris kifejezés: '[a-z&&^c-f]'	
Illeszkedésre példa	**Nem illeszkedő példa**
a	D
z	A

4. Előre definiált karakterosztályok

A gyakorlatban a gyakran használt karakterosztályok halmaza meglehetősen szűk (betűk, számok, stb.), ezeknek nagy része a nyelvben beépítve is megtalálható.
Beépített karakterosztályok[41]:

- . - bármilyen karakter
- *d* – decimális karakterek ([0-9])
- *D* – nem decimális karakterek ([^0-9])
- *s* – whitespace karakterek ([\n\t\r\f])
- *S* – nem whitespace karakterek ([^\s])
- *w* – latin ábécé szó karakterei ([a-zA-Z0-9_]), ékezetes betűk nem
- *W* – nem szó karakterek ([^\w])
- *p{Ll}* – bármilyen kisbetű, ideértve minden ékezetest is
- *p{Lu}* – bármilyen nagybetű, ideértve minden ékezetest is
- *p{L}* – bármilyen betű, ideértve minden ékezetest is

Ha a keltezésnél nem vagyunk biztosak az évben, de tudjuk, hogy 1000 utáni, decemberi dátumról van szó, akkor a kifejezésünk a következőképpen is írható:

[40] Unió képzésnél lehetőség van a karakterosztályok leírásának egyszerűsítésére. Azaz az egymásba ágyazott karakterosztályoknál a belső definiáló szögletes zárójelek elhagyhatók. Például a '[[a]|[d-e]]' írható '[ad-e]'-nek is. Ez kizárásra, illetve metszetre nem vonatkozik.
[41] Információk az UNICODE támogatásáról: http://www.regular-expressions.info/unicode.html

Reguláris kifejezés: '[BP]écs\, [1-9]\d\d\d\. december \d\d\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Pécs, 978. december 07.
Pécs, 1978. december 09.	Pécs, 1978. december 7.
Pécs, 2978. december 09.	Mécs, 2978. december 09.

5. Karaktercsoportok

Lehetőség van arra, hogy a mintán belül csoportokat hozzunk létre. A csoport olyan mintarész, melyet egy nyitó és az ahhoz tartozó csukó zárójel határol. A mintán belüli csoportok számozottak. A nullás az egész minta, az n-edik pedig az a csoport melynek nyitózárójele az n-edik nyitózárójel a minta elejéről nézve.

- Reguláris kifejezés: ([BP]écs)\, (([1-9]\d\d\d)\. december \d\d\.)
- Egyes csoportok:

0. csoport: ([BP]écs)\, (([1-9]\d\d\d)\. (december) (\d\d)\.)
1. csoport: ([BP]écs)
2. csoport: (([1-9]\d\d\d)\. december \d\d\.)
3. csoport: ([1-9]\d\d\d)

Az egyes csoportok a reguláris kifejezésen belül hivatkozhatók. Ilyenkor a hivatkozott csoportra illeszkedő szövegrésznek meg kell ismétlődnie a mintában a hivatkozás helyén is.
Szintaxis: *(<minta>)....\<minta azonosító>*

Reguláris kifejezés: '(\w)vs\.\1'	
Illeszkedésre példa	**Nem illeszkedő példa**
1vs.1	avs.b
3vs.3	2vs.1

6. Csoportok ismétlése

Az eddig ismertetettek lehetővé teszik a minta egyes pozícióiban elfogadható karakterek igen pontos leírását. Ellenben arra még nem adnak módszert, hogy a mintában előforduló típus ismétléseket hogyan kezeljük.
Ismétlés megadásának szintaxisa:

- *<karakterosztály vagy karakter csoport>{<n>}* - pontosan 'n'-szeres ismétlés a mintának
- *< karakterosztály vagy karakter csoport>{<n, m>}* - 'n' és 'm' közé esik az ismétlések száma ('n' és 'm' ismétlés megengedett)
- *< karakterosztály vagy karakter csoport>{<n, >}* - legalább 'n' ismétlés
- *< karakterosztály vagy karakter csoport>+* - ekvivalens a {1,}-gyel
- *< karakterosztály vagy karakter csoport>?* - ekvivalens a {0,1}-gyel
- *< karakterosztály vagy karakter csoport>** - ekvivalens a {0,}-gyel

A előző példában várhatóan bármely helységet elfogadjuk (a helységről tételezzük fel, hogy nagybetűvel kezdődik, amelyet legalább egy kisbetű követ). Mindezek alapján a mintánk következőképpen alakul:

Reguláris kifejezés: '\p{Lu}\p{Ll}+, [1-9]\d{3}\. \p{Ll}+ \d{2}\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Szeged, 1978. december 07.	Szeged, 978. december 07.
Budapest, 1978. december 07.	nappalodott, 978. december 07.

Ezzel azonban még nem definiáltuk, hogy ha ismétléseket tartalmazó minta többféleként is illeszkedhet, akkor melyiket szeretnénk használni. A két legfontosabb kiértékelési mód a mohó és a lusta.

6.1. Mohó kiértékelés

Ilyenkor az ismételhető minta, részminta a lehető legtöbb elemre próbál illeszkedni. Ha ez nem lehetséges, mert így az egész mintának nincs illeszkedése, akkor az utolsó lehetséges karaktert kivéve mindegyikre próbál illeszkedik. És így tovább. Ez a mintaillesztés alap viselkedése.

Reguláris kifejezés: '(.+)(\d{1,2})\.'		
Illesztett szöveg: Pécs, 1978. december 07.		
Illesztett szöveg (0. csoport)	**1. csoport**	**2.csoport**
Pécs, 1978. december 07.	Pécs, 1978. december 0	7

6.2. Lusta kiértékelés

A mohó fordítottja. Elsőként a lehető legkevesebb elemet felhasználva próbálja a részmintákat illeszteni. Csak ha ez nem lehetséges, akkor bővíti az illeszkedés hosszát.
Szintaxis: <ismételt csoport>?

Reguláris kifejezés: '(.+?)(\d{1,2})\.'		
Illesztett szöveg: Pécs, 1978. december 07.		
Illesztett szöveg (0. csoport)	**1. csoport**	**2.csoport**
Pécs, 1978. december 07.	Pécs, 1978. december	07

7. További hasznos funkciók

Bizonyos körülmények között több alapvetően eltérő mintát is el kell fogadnunk. A korábbi keltezéses példánál maradva elképzelhető, hogy a keltezés helye egyszerűen ki van pontozva. Az ilyen problémák megoldására javasolt a minták vagylagos összefűzése.
Szintaxis: *(<első minta>)|(<második minta>)*

Reguláris kifejezés: '(\p{Lu}\p{Ll}+, [1-9][\d]{3}\. \p{Ll}+ [\d]{2}\.)|(\.{10,})'	
Illeszkedésre példa	**Nem illeszkedő példa**
Szolnok, 1978. december 07.	reggeledett, 1978. december 07.
.....	....

A keltezés tulajdonsága, függően a használt nyelvtől, hogy csak a sor elején vagy mindig egy sor lezárásaként szerepel. Erre szolgáló beépített módosítók:

- Sor elejére illeszkedik: ^
- Sor végére illeszkedik: *$*

Reguláris kifejezés[42]: '^(\p{Lu}\p{Ll}+, [1-9][\d]{3}\. \p{Ll}+ [\d]{2}\.)|^(\.{10,})'	
Illeszkedésre példa	**Nem illeszkedő példa**
Szolnok, 1978. december 07.	Kelt.: 978. december 07.
.....	Kelt.: .....

A Perl reguláris kifejezéseknél lehetőség van tovább speciális opciók bekapcsolására. Ezen opciók közül a gyakorlatban a legfontosabb a kisbetű-nagybetű érzékenység kikapcsolása. Ez az opció annak a csoportnak a hátralevő részéig fejti ki hatását ahol definiálva lett.
Szintaxis: (?i)<reguláris kifejezés>

Reguláris kifejezés: '([A-Z](?i)[a-z]+e)[a-z]'	
Illeszkedésre példa	**Nem illeszkedő példa**
Losangeles	LOSANGELES
LosAngeles	losAngeles
LOSAngelEs	Losangeles

A Perl szintaxis lehetővé teszi kommentek elhelyezését a reguláris kifejezésben[43].
Szintaxis: (?#<komment szövege>)

[42] Itt a problémát, mint részminta keresést vizsgáljuk.
[43] A Java reguláris kifejezés motorja nem támogatja.

Reguláris kifejezés: 'júli(?#ez a reguláris kifejezés júliusra illeszkedik)us'	
Illeszkedésre példa	**Nem illeszkedő példa**
Július	júli(#ez a reguláris kifejezés júliusra illeszkedik)us'

8. Reguláris kifejezés alapú DoS támadás

A reguláris kifejezések, minden hasznuk mellett is, bizonyos körülmények között veszély forrásai lehetnek. Egyes esetekben más sebezhetőségek könnyebb kihasználhatóságát teszik lehetővé [R6], máskor nem megfelelő alkalmazásuk képezi a problémát [R7]. Jelen jegyzet szempontjából az utóbbi eset, annak is a weblapok bemeneti adatinak reguláris kifejezésekkel történő ellenőrzését kihasználó úgynevezett REDoS (R*egular* E*xpression* D*enial of* S*ervice*) támadás bír kiemelt fontossággal.
A REDoS támadás azt az igen elterjedt és helyes gyakorlatot használja ki, hogy az egyes honlapok a bemeneti adataik formátumának ellenőrzésére reguláris kifejezéseket alkalmaznak – ilyen adat lehet például a felhasználó neve vagy éppen a születési éve.
Példaként vizsgáljuk meg a következő kifejezést, melynek célja a megjegyzések ellenőrzése lenne: ’^(\p{L}+\s?)+$’. A kifejezés azokra a szövegekre illeszkedik, amelyek egy vagy több, egymástól whitespace karakterrel elválasztott szóból állnak. Nézzük, hogy a következő szövegekre hányféleképpen illesztethető:

- ’a’: 1
- ’aaaa’: 16
- ’aaaaaaaaaaaaaaaa’: 65536

Látható, hogy a szöveg hosszával nem lineárisan, hanem exponenciálisan növekszik a lehetséges illeszkedések száma. A problémát elsősorban az eset képezi, amikor végül nem találunk illeszkedést, mint a ’aaaa-’. Itt a reguláris kifejezés értelmezőnek az ’aaaa’ mind a 16 lehetséges felosztását ki kell próbálnia mielőtt, kijelenthetné: nincs illeszkedés (elvileg a 16-ból bármelyiknél lehetne szerencséje, azaz illeszkedést lelhetne).
Az ilyen exponenciális robbanást produkáló reguláris kifejezések okozta sebezhetőség alkalmas lehet a kiszolgáló leterhelésére, azaz DoS támadás megvalósítására. Természetesen a támadást jelentősen megnehezíti, hogy minden reguláris kifejezéshez külön ki kell találni egy olyan bemenetet, amely mellett az exponenciális robbanás bekövetkezik.
Veszélyes reguláris kifejezések jellemző felépítése:

- Tartalmaz karaktercsoport ismétlést
- Az ismételt csoporton belül ismétlést (például: ’(a+)+’) vagy átfedő alternatívákat (például: ’(a|aa)+’)

9. Java által biztosított reguláris kifejezés API[44]

A Java 1.4-es változatába kerültek bele a Pattern, illetve a Matcher osztályok.

9.1. Java.util.regex.Pattern[45]

A Pattern osztály felelős a reguláris kifejezések feldolgozásáért, illetve az egyszerű mintailleszkedés ellenőrzésért.
Fontosabb metódusai:

- *public static boolean* ***matches(String reg, CharSequence input):*** Igaz, ha a reg kifejezés illeszkedik a bemenetként megadott teljes szövegre.
- ***public static Pattern compile(String regex)***: A regexp paraméterként kapott kifejezést lefordítja, az alapján készít egy Pattern objektumot, ami a továbbiakban illeszkedés vizsgálatra használható.

[44] Mivel a Java nyelvben a String definíciója során a '\' jel speciális karakter, ezért az előre definiált reguláris kifejezésekben '\\'-ként írandó. Hasonlóan, a dupla visszapert négy egymást követő visszaperjellel lehet leírni, ami az illeszkedésben egy visszaperjel meglétét fogja megkövetelni.
[45] http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

- ***public String[] split(CharSequence input)***: A korábban megadott reguláris kifejezés illeszkedései közötti szövegrészeket adja vissza.
- ***public Matcher matcher(CharSequence input)***: Az inputra vonatkozó Matcher típusú objektummal tér vissza, ami lehetővé teszi a további, hatékony művelet végzést.

9.2. Java.util.regex.Matcher[46]

A Matcher feladata az egyes illeszkedések lekérdezése, manipulálása a Patternek átadott karaktersorozatban.
Fontosabb metódusai:

- *public int **end()***: A jelenleg vizsgált illeszkedés utáni első karakter pozícióját adja vissza.
- *public int **end**(int group)*: A group által azonosított karaktercsoport utolsó illeszkedő elemére következő karaktert adja vissza, -1-t ha az adott csoport nem illeszkedett.
- *public String **group()***: Az illeszkedés szövegét adja vissza.
- *public String **group**(int group)*: A group által jelölt karaktercsoport aktuális illeszkedő szövegét adja vissza.
- *public boolean **find()***: Megpróbálja megkeresi a minta következő illeszkedését a szövegben. Igazat ad vissza, ha van még illeszkedés, egyébként hamisat.
- *public boolean **lookingAt()***: Igaz, ha a reguláris kifejezés illeszthető a szövegre vagy annak egy részére.
- *public boolean **matches()***: Igaz, ha a reguláris kifejezés illeszthető az egész szövegre.
- *public String **replaceAll**(String replacement)*: Lecseréli a reguláris kifejezés összes illeszkedését a replacment-ben megadott kifejezés alapján. A replacment tartalmazhat hivatkozásokat a reguláris kifejezés egyes karaktercsoportjaira.

Ajánlott irodalom:

A1. Jeffrey E. F. Friedl: Reguláris kifejezések mesterfokon

A2. Laura Lemay: Perl mesteri szinten 21 nap alatt

A3. http://java.sun.com/docs/books/tutorial/essential/regex/index.html

Referencia:

R1. http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

R2. http://www.perl.com/doc/manual/html/pod/perlre.html

R3. http://msdn.microsoft.com/en-us/library/hs600312%28VS.71%29.aspx

R4. http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/package-summary.html

R5. http://hu.php.net/manual/en/book.pcre.php

R6. http://www.ihteam.net/papers/blind-sqli-regexp-attack.pdf

R7. http://msdn.microsoft.com/en-us/magazine/ff646973.aspx

10. Melléklet

A példák megértését, illetve a reguláris kifejezések írását megkönnyítendő, mellékletként egy Java alkalmazást teszünk közre, mely a reguláris kifejezések tesztelésére használható.

10.1. Fordítás, futtatás

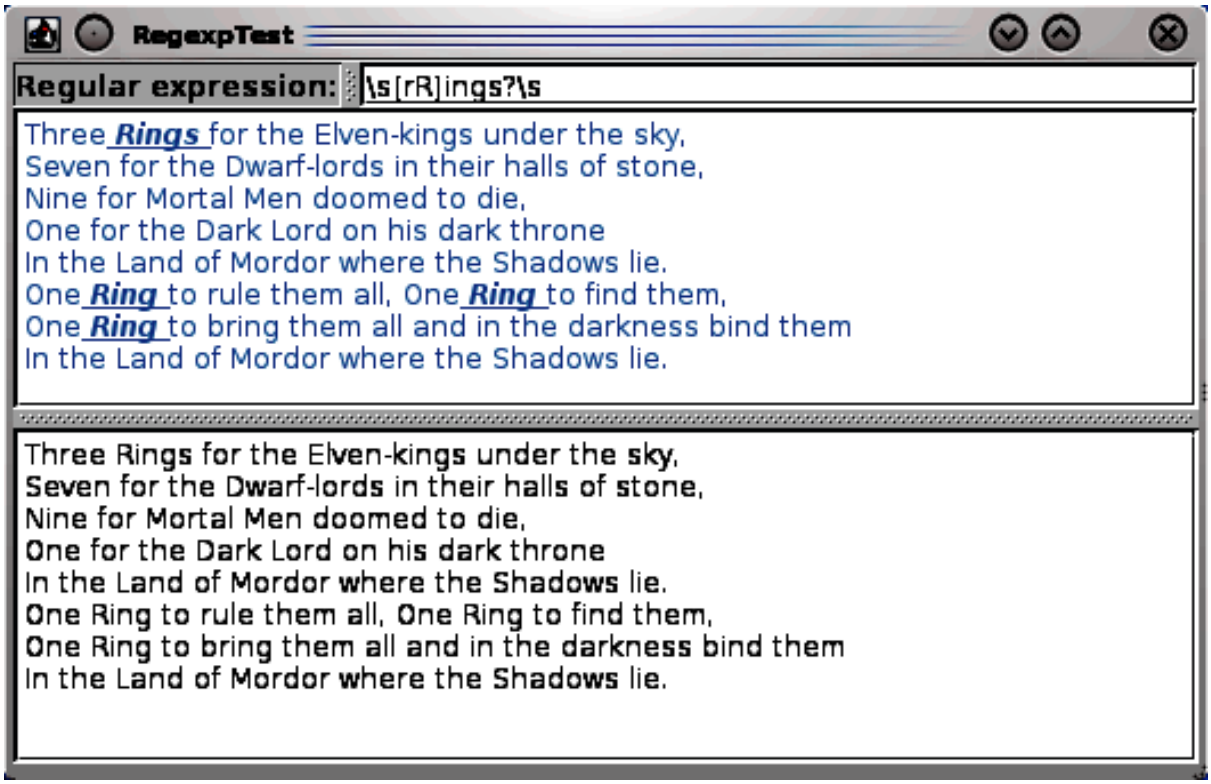
A teszt alkalmazás fordításának, futtatásának lépései:

[46] http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Matcher.html

1. Legalább 1.4-es Java JDK telepítése a számítógépre
2. A melléklet végén található kód kimásolása egy *RegexpTest.java* nevű fájlba
3. A program az előző pontban létrehozott könyvtárban a következő utasítással fordítható: *javac -g RegexpTest.java*
4. Futtatás: *java RegexpTest*

10.2. Használat

Tételezzük fel, hogy szeretnénk megkeresni a Gyűrűk Ura című könyvben megtalálható, a gyűrűk szétosztását leíró versben a 'ring' szó összes előfordulását.
A vizsgálat elvégzéséhez indítsuk el az alkalmazást. Az ablak jobb felső sarkában lévő szöveg mezőbe kell beírni azt a reguláris kifejezést, amelynek az illeszkedéseit keressük (az esetünkben: '\s[rR]ings?\s'). Az alsó mezőbe kerül a szöveg, melyben az illeszkedéseket vizsgáljuk. A szoftver az alsó mező tartalmát automatikusan a felsőbe másolja és ott az illeszkedő elemeket félkövér, dőlt, aláhúzott karakterrel jeleníti meg, lásd 1. ábra. Ezek alapján a 'ring' szó négyszer szerepel a versben.



1. ábra

10.3. Alkalmazás kódja

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JSplitPane;
import javax.swing.JLabel;
import javax.swing.JTextPane;
import javax.swing.text.SimpleAttributeSet;
```

```
import javax.swing.text.StyleConstants;
import javax.swing.JScrollPane;

public class RegexpTest extends JFrame {

	private static final long serialVersionUID = 1L;

	// UI components
	private JPanel jContentPane = null;
	private JSplitPane jSplitPane = null;
	private JLabel regexpLabel = null;
	private JTextField regexpTextField = null;
	private JSplitPane splitPane = null;
	private JTextPane sourceTextPane = null;
	private JTextPane matchTextPane = null;
	private JScrollPane upperScrollPane = null;
	private JScrollPane lowerScrollPane = null;

	// Pattern matching objects
	private Pattern pattern = null;
	private Matcher matcher = null;
	private SimpleAttributeSet matchedTextAttributes = null;

	/**
	 * Creats the new pattern object from the input field's content if possible.
Otherwise
	 * pattern is set to null
	 */
	private void generatePattern()
	{
		try
		{
		pattern = Pattern.compile(regexpTextField.getText());
		}
		catch(PatternSyntaxException e)
		{
			pattern = null;
			matcher = null;
		}
	}

	/**
	 * Shows the matches of the pattern in the upper pane
	 */
	private void showMatches()
	{
		matchTextPane.setText(sourceTextPane.getText());
		if (pattern == null)
		{
			return;
		}

		matcher = pattern.matcher(sourceTextPane.getText());

	  while (matcher.find()) {
		matchTextPane.setSelectedTextColor(Color.WHITE);
		matchTextPane.setSelectionStart(matcher.start());
		matchTextPane.setSelectionEnd(matcher.end());
		matchTextPane.setSelectionColor(Color.WHITE);

	matchTextPane.getStyledDocument().setCharacterAttributes(matcher.start(),
				matcher.end() - matcher.start(), matchedTextAttributes,
true);
	  }
	}

	/**
	 * Generates the text style used to mark matches
```

```
	 */
	private void generateMatchedTextAttributes() {
		// Style of the matched text
	    matchedTextAttributes = new SimpleAttributeSet();

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Bold,
Boolean.TRUE);

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Italic,
Boolean.TRUE);

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Underline,
Boolean.TRUE);
	}

	public RegexpTest() {
		super();
		initialize();
		generateMatchedTextAttributes();
	}

	private void initialize() {
		this.setSize(400, 400);
		this.setContentPane(getJContentPane());
		this.setTitle("RegexpTest");
	}

	/**
	 * UI components
	 */

	private JPanel getJContentPane() {
		if (jContentPane == null) {
			jContentPane = new JPanel();
			jContentPane.setLayout(new BorderLayout());
			jContentPane.add(getJSplitPane(), BorderLayout.NORTH);
			jContentPane.add(getSplitPanel(), BorderLayout.CENTER);
		}
		return jContentPane;
	}

	private JSplitPane getJSplitPane() {
		if (jSplitPane == null) {
			regexpLabel = new JLabel();
			regexpLabel.setText("Regular expression:");

			jSplitPane = new JSplitPane();
			jSplitPane.setLeftComponent(regexpLabel);
			jSplitPane.setRightComponent(getRegexpTextField());
			jSplitPane.setEnabled(false);
		}
		return jSplitPane;
	}

	private JTextField getRegexpTextField() {
		if (regexpTextField == null) {
			regexpTextField = new JTextField();
			regexpTextField.setToolTipText("Regular expression");
			regexpTextField.addKeyListener(new java.awt.event.KeyAdapter()
{
				public void keyReleased(java.awt.event.KeyEvent e) {
					generatePattern();
					showMatches();
				}
			});
		}
		return regexpTextField;
	}
```

```
	private JSplitPane getSplitPanel() {
		if (splitPane == null) {
			splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
			splitPane.setTopComponent(getUpperScrollPane());
			splitPane.setBottomComponent(getLowerScrollPane1());

		}
		return splitPane;
	}


	private JTextPane getSourceTextPane() {
		if (sourceTextPane == null) {
			sourceTextPane = new JTextPane();
			sourceTextPane.setToolTipText("Test text");
			sourceTextPane.addKeyListener(new java.awt.event.KeyAdapter() {
				public void keyReleased(java.awt.event.KeyEvent e) {
					showMatches();
				}
			});
		}
		return sourceTextPane;
	}

	private JTextPane getMatchTextPane() {
		if (matchTextPane == null) {
			matchTextPane = new JTextPane();
			matchTextPane.setEnabled(false);
			matchTextPane.setDragEnabled(false);
			matchTextPane.setFocusable(false);
			matchTextPane.setToolTipText("Matched words are in italic, bold
and underlined");
		}
		return matchTextPane;
	}

	private JScrollPane getUpperScrollPane() {
		if (upperScrollPane == null) {
			upperScrollPane = new JScrollPane();
			upperScrollPane.setViewportView(getMatchTextPane());
		}
		return upperScrollPane;
	}

	private JScrollPane getLowerScrollPane1() {
		if (lowerScrollPane == null) {
			lowerScrollPane = new JScrollPane();
			lowerScrollPane.setViewportView(getSourceTextPane());
		}
		return lowerScrollPane;
	}

	public static void main(String[] args)
	{
		RegexpTest mainclass = new RegexpTest();
		mainclass.setVisible(true);
	}
}
```

IV. Függelék: Webes és adatbázis biztonsági kérdések

Szerzők: Balázs Zoltán, Paksy Patrik, Sallai Tamás, Veres-Szentkirályi András

1. Bevezetés

A dinamikus weboldalak és a mögöttük lévő adatbázisokban tárolt adatok manapság kiemelt támadási célpontnak számítanak, ezért az adatok védelme kritikus feladat. Sok esetben az adatok bizalmasságának és sértetlenségének biztosítására nem fektetnek nagy hangsúlyt, pedig néhány egyszerű módszerrel már igencsak megnehezíthetjük a támadók munkáját.

A segédlet célja, hogy rámutassunk azokra a főbb problémákra, biztonsági kérdésekre és a hozzájuk tartozó védekezési lehetőségekre, melyekkel dinamikus weboldalak készítése során adatbázis adminisztrátorként vagy fejlesztőként találkozhatunk. Elsőként megemlítünk néhány fontos szempontot, melyeket az adatbázis konfigurálásánál érdemes betartanunk, majd pedig bevezető jelleggel tárgyalunk három gyakori alkalmazás sebezhetőséget, az SQL injection, XSS, és CSRF támadásokat. A támadási felületet elsősorban a bemeneti interfészek biztosítják, így kiemelten foglalkozunk az input ellenőrzésének kérdésével. Fejlesztőként elengedhetetlen, hogy ezekkel tisztában legyünk, és megtegyük a legalapvetőbb lépéseket ahhoz, hogy bizalmas adatok ne kerüljenek illetéktelen kézbe, vagy jogosulatlanul ne lehessen elvégezni bizonyos kéréseket. Minden esetben figyelembe kell vennünk azt a tényt is, hogy a védelmi intézkedéseink mértéke akkor megfelelő, ha arányban van a védendő adatok értékével, de az itt ismertetett módszerek egy jó webes alkalmazásból gyakorlatilag ma már nem hiányozhatnak.

A dokumentum elsősorban áttekintést próbál adni az említett témakörökről. A bővebb információk után érdeklődőknek a segédlet végén található hivatkozások nyújtanak további segítséget.

2. Konfigurációs védelmi szempontok

2.1. Adatbázis jogosultságok

Az egyik legfontosabb biztonsági alapelv, hogy minden program vagy felhasználó a szükséges lehető legkevesebb joggal rendelkezzen. Ez egyrészt igaz az adatbázis-kezelő futtatására (vagyis ne rootként fusson az adatbázis-kezelő), másrészt igaz a webszerver és adatbázis-kezelő kapcsolatára. Bár a fejlesztés során mindig kényelmes, ha nem ütközünk jogosultsági hibába, azonban az éles működés során már tilos a sémafelhasználó vagy

adatbázis-adminisztrátori jogokkal csatlakozni a webszerverről az adatbázishoz. Mindig külön felhasználót (ún. proxy felhasználó) hozzunk létre erre a feladatra, amely csak és kizárólag a szükséges funkciókhoz elegendőek. Így például, ha egy támadó a felhasználó nevében megkerüli a webalkalmazásba épített biztonsági logikát, de ezen felhasználónak csak lekérdezési joga van, akkor a támadó nem lesz képes a különböző adatmódosító utasítások (DML) végrehajtására.

2.2. Puffer túlcsordulás

Mivel az Oracle adatbázis-kezelő core funkcióit C nyelvben írták, ezért szinte természetes, hogy puffer-túlcsordulás alapú hibák is előfordulnak benne. Ezeket a hibákat az Oracle adatbázis biztonsági frissítéseivel lehet javítani.

2.3. Többszintű védekezés – layered security, defense in depth

Gyakori téves feltételezés, hogy mivel az adatbázist tűzfallal szeparálják az internettől, ezért az adatbázis védelmére nem szükséges kellő hangsúlyt helyezni, így például alapértelmezett felhasználói jelszavakat hagynak bent, melyek könnyű hozzáférést adnak a támadók kezébe.
Másik tipikus hiba, hogy az alkalmazások kompatibilitása miatt (vagy egyszerűen lustaságból) nem frissítik biztonsági frissítésekkel az adatbázis-kezelőket vagy akár olyan csomagok is telepítésre kerülnek, amelyek használatát semmi sem indokolja. Ezek olyan sérülékenységet tartalmazhatnak, amelyek kihasználásával jogosulatlan kódvégrehajtás történhet.

3. Alkalmazások biztonsági kérdései

3.1. SQL kódbeillesztés (SQL injection)

Áttekintés

A legfontosabb biztonságot növelő módszer a már szóba került input validáció. Bármely interfészen érkező adatot megbízhatatlanként kell kezelni mindaddig, amíg bizonyos ellenőrzéseken (pl. tartomány illetve értékkészlet vizsgálat, reguláris kifejezések) sikeresen át nem esett az adat. Biztonsági és teljesítmény szempontból is rossz megközelítés, amikor megpróbáljuk felsorolni a rossz, hibás bemeneteket, és ezeket blokkoljuk (black list alapú megközelítés), hiszen az ilyen listák sosem teljesek, és folyamatosan frissíteni kell.
Az ellenőrzés elmulasztása akár egy bemeneti paraméteren is, komoly biztonsági következményekkel járhat. Egy támadónak ilyenkor lehetősége nyílik arra, hogy a fejlesztő által megírt SQL kódot tetszőleges kóddal kiegészítse, és azt a felhasználó jogosultságaival futtassa. Így a támadónak lehetősége nyílhat a felhasználói bejelentkezés megkerülésére, a teljes adatbázis lemásolására, új adatbázis-adminisztrátor létrehozására, operációs rendszer parancsok végrehajtására vagy tetszőleges kód feltöltésére – egyetlen nem ellenőrzött paraméter miatt. Adatbázis-kezelő, illetve szerver-oldali programozási nyelvfüggő, hogy az SQL kódbeillesztéses támadás során csak a már elkezdett utasítást tudjuk folytatni, vagy lehetőségünk van több parancs egymás utáni végrehajtására is – egyetlen webszerver kéréssel. Az utóbbit kód stackingnek nevezzük, melyet az Oracle nem támogat, így csak azt az utasítást tudjuk kiegészíteni, amelyiket a program elkezdte.
Tipikus tévhit, hogy ha egy paraméter legördülő menü, rádiógomb formájában szerepel a weboldalon, vagy ha kliensoldali JavaScript ellenőrzést végeztünk a beviteli mezőn, akkor azon értékek már megbízhatónak tekinthetőek. A szerveroldali input validáció kötelező eleme minden webes alkalmazásnak, míg a kliensoldali JavaScript ellenőrzés elsősorban nem biztonsági szerepet játszik, hanem a felesleges űrlapküldések elkerülésére, ezzel együtt a szerver terhelésének csökkentésére valamint a felhasználók kényelmére szolgál. Az input

validáción kívül kiemelten fontos az output validáció[47] is, az ehhez kapcsolódó támadásokról a későbbi fejezetekben is olvashatunk.

Hibakezelés

Szintén alapelv, hogy éles alkalmazás esetén ne adjunk vissza olyan hibaüzenetet a felhasználónak, amiből következhet az adatbázis vagy az alkalmazás felépítésére. Például egy éles alkalmazásban egy hibás SQL lekérdezés esetén a helyes megjelenítendő hibaüzenet: „Belső hiba történt”, míg a kerülendő hibaüzenet típusok: „ORA-01790: expression must have same datatype as corresponding expression”. Az utóbbi típusú üzenetek sokat segítenek a támadónak abban, hogy az SQL kódbeillesztéshez érvényes SQL kódot készítsen.
PHP-ban például az error_reporting(0); paranccsal tudjuk forráskódból kikapcsolni a hibaüzeneteket, majd ha a lekérdezés visszatérési értéke üres, akkor lehet egy általános hibát megjeleníteni. Ha nem kapcsolnánk ki minden hibaüzenetet, PHP-ban van lehetőség egy adott parancs futtatásakor keletkező hibának elnyomására is, ehhez a parancs elé az „@” jelet kell beírni.

SQL kódbeillesztés példák

A következő két példa szemléltetésként szolgál az SQL kódbeillesztés alapú támadásokhoz. Természetesen rengeteg más típus is létezik, most két egyszerű módszert mutatunk be.
Példa I. (mindig igaz WHERE feltétel konstruálása)
Tekintsünk egy olyan weboldalt, ahol nem kezelték megfelelően a kódbeillesztési támadásokat. Az oldalon egy bejelentkezési űrlap található, melynek elküldésekor az alábbi SQL kód fut le. Ha találunk az adatbázisban a megadott paraméterekkel felhasználót, tehát a lekérdezés nem nulla sorral tér vissza, akkor sikeres volt a bejelentkezés.

```
SELECT * FROM felhasznalok
WHERE nev = '$nameField' AND jelszo = '$pwdField';
```

A kódbeillesztés fontos lépése, hogy a megfelelő szintaxis megtartásával biztosítsuk a kód lefutását. Ezért figyelnünk kell arra, hogy a megkezdett aposztrófokat megfelelően zárjuk le, a felesleges SQL kódrészeket pedig kommentezzük ki. Az utasítás hátralévő részének kikommentezésére a „--” jel való.

Beillesztett SQL kódrészlet: Teszt' OR 1 = 1; --

```
SELECT * FROM felhasznalok
WHERE nev = 'Teszt' OR 1 = 1; --' AND jelszo = 'TesztJelszo';
```

A beillesztett kóddal a következő lekérdezés fog lefutni, így a bejelentkezés sikeres lesz:

```
SELECT * FROM felhasznalok
WHERE nev = 'Teszt' OR 1 = 1;
```

Példa II. (UNION)
Kódbeillesztésnél gyakran használt módszer, amikor a lekérdezéshez az UNION művelet segítségével kapcsolunk hozzá egy másik lekérdezést, ezáltal kinyerhetjük egy tetszőleges tábla tartalmát. A sikeres támadáshoz az UNION két tulajdonságát kell felhasználni: ugyanannyi mezőt kell megadnunk a SELECT záradékban mindkét lekérdezésnél, és a mezőknek páronként azonos típusúnak kell lenniük (a *null* érték minden típussal azonos).
Tegyük fel, hogy egy oldalon az URL-ben megadott ID-val rendelkező hírt jelenítjük meg az alábbi lekérdezéssel:

[47] Output validáció: https://www.owasp.org/index.php/Output_Validation

```
SELECT cim, szoveg FROM hirek
WHERE id = $urlParam;
```

Célunk pedig a jelszavak listájának megszerzése a *felhasznalok* táblából:

```
SELECT cim, szoveg FROM hirek
WHERE id = 1 UNION SELECT null, jelszo FROM felhasznalok;
```

Beillesztett kódrészlet: 1 UNION SELECT null,jelszo FROM felhasznalok

Védekezés

A legegyszerűbb védekezési módszer, ha escape-eljük a speciális karaktereket, így az SQL injection megvalósításához szükséges aposztróf vagy kötőjel nem speciális jelentéssel bíró szimbólumként értelmeződik. Az escape-elés során minden speciális karakter elé egy „\” jel kerül. Mivel ezek a karakterek többféle formában megadhatók, így önmagában nem mindig nyújt megfelelő védelmet.
Adatkötés használata esetén a lekérdezést paraméteresen, helyőrzőkkel adjuk meg, az adatbázis-kezelő pedig a paraméterek helyét kihagyva készíti el a végrehajtási tervet. Így a behelyettesítés során már nem változhat meg a lekérdezés szerkezete, az esetleges rosszindulatú kód paraméterként fog szerepelni a lekérdezésben, nem pedig SQL kódként. A PHP változókat az oci_bind_by_name() metódus segítségével köthetjük a helyőrzőkhöz.
Példa (adatkötés):

```
$sqlQuery = "SELECT * FROM table WHERE id = :id";
$statement = oci_parse($sqlQuery);

oci_bind_by_name($statement, ":id", 12345);
oci_execute($statement);
```

Ha nincs lehetőség paraméterezett SQL kódra, akkor használjunk tárolt eljárásokat. Tárolt eljárásnál kerüljük a dinamikus SQL használatát, vagy használjunk adatbázisoldali input-validációt (pl. DBMS_ASSERT Oracle esetében).

3.2. XSS

Áttekintés

A Cross-Site Scripting (XSS) támadás során a támadó JavaScript kódot injektál az áldozat gépén a megtámadott weboldalba. Ez azért lesz veszélyes, mert ez a script az oldalon majdnem mindenhez hozzáfér, tudja manipulálni, hogy mi jelenjen meg és kifele kapcsolatokat is nyithat. Általában a weblapok a saját domainjükből jövő tartalmakban megbíznak, más domainből jövőekben pedig nem (ez a Same Origin Policy: ami onnan jött ahonnan az oldal, az szabad kezet kap mindenhez, ami máshonnan, az csak nagyon kevés dolgot tehet meg).
Általában feltételezzük, hogy minden támadáshoz az áldozat rákattint egy linkre, ami a támadótól származik.

A támadás megvalósítása

A támadás alapja, hogy a támadó által megírt JavaScript kód valamilyen formában az áldozathoz kerüljön a weblap által. Ehhez képzeljünk el egy esetet, amikor a weblapon lehet keresni, és a kereső (mint általában szokás) a keresési eredmények fölé kiírja a keresett kifejezést. Ezt szerveroldalon egy egyszerű String összefűzéssel éri el, tehát ami a klienstől jött, az egy az egyben kikerül a kimenetre is.
Például:
Eset I., a megfelelő működés:

- URL: localhost/?search=kif
- HTML forrás: Keresés:kif
- Eredmény: **Keresés: kif**

Eset II., a támadás:

- URL: localhost/?search=kif<script>alert('xss')</script>
- HTML forrás: Keresés:kif<script>alert('xss')</script>
- Eredmény: **Keresés: kif**, valamint felugrik egy popup az xss szöveggel, tehát sikeresen kódot injektáltunk a weboldalba.

Tehát ha az áldozat rákattint a támadó által küldött linkre, akkor a támadó által megírt JavaScript lefut.

Perzisztens és reflektív támadás

Az előző pontban bemutatott megoldás az ún. reflektív támadás, mivel kizárólag a kérésben szerepel a támadó kód. A másik változat a perzisztens XSS, amely úgy futtat kódot az áldozatnál, hogy azt a webszerver tárolja. Ez általában azért lehet veszélyesebb, mert a megtámadott weblap összes látogatóját érintheti.
A perzisztens eset is arra épül, hogy a felhasználótól jövő bemenet módosítás nélkül kikerül az oldalra, csak ebben az esetben valamilyen eltárolt adat által. Ez lehet pl. egy komment egy blogbejegyzéshez, ahova regisztrált vagy anonim felhasználó szabad szöveget írhat.
Például: A weboldalon van egy kommentelési lehetőség, és ezek a bejegyzés alatt megjelennek. Ezután a támadó véleménye ez a cikkről:

```
Nagyon jó, csak így tovább!<script>alert('xss')</script>
```

Ezután bárki megnézi a bejegyzést, a Javascript kód le fog futni nála.

Veszélyesség

Persze egy alert feldobása miatt még nem lenne olyan veszélyes, azonban a Javascript nagyfokú szabadsága miatt változatos (és nem feltűnő!) módon lehet kihasználni az XSS sebezhetőségeket.
A támadó legfontosabb célja a SessionID megszerzése lehet, mivel ha azt sikerül megtudnia, akkor egyszerűen meg tudja személyesíteni az áldozatot. A támadás vázlata az alábbi lehet:

- Az áldozat rákattint a linkre
- Javascript kiolvassa a SessionID-t a sütik közül
- Nyit egy kérést egy, a támadó által irányított gépre, amiben elküldi a SessionID-t
- A támadó a saját SessionID-jét átírja az imént megszerzettre
- Megnyitja a támadott weboldalt, és az áldozat nevében van bejelentkezve

Mivel a DOM-ot és a futtatott Javascripteket is tetszés szerint tudja módosítani, ezért nem nehéz a bejelentkező formot átírnia olyanra, hogy mellékesen a beírt adatokat még a támadó által irányított gépre is elküldje.
Mivel a megjelenítést is tudja módosítani, ezért megváltoztathatja a weboldalon közölt tartalmakat, pl. egy tőzsdeoldalon nem valós árfolyamadatokat mutat, ezzel befolyásolhatja a piaci folyamatokat.
Különösen veszélyes, hogy egy rejtett (1×1-es méretű, keret nélküli) IFrame-be (Inline Frame) be tudja tölteni a weboldalt, és ahhoz szabadon hozzá tud férni. Ez azért lehetséges, mert ugyanarról a domainről lett betöltve, ezért a böngésző megbízhatónak tekinti a kódot.

Védekezés

A legfontosabb védekezés, hogy minden felhasználói bevitt adatot gonosznak tekintünk és validálunk, valamint minden dinamikus adatot amit megjelenítünk escape-lünk. Ez utóbbi a speciális karakterek lecserélését jelenti, így a böngésző biztosan nem fogja azokat végrehajtani.
Az előbbi pedig az esetek többségében whitelistinget jelent, tehát csak azokat a bemeneteket fogadjuk el, ami illeszkedik egy bizonyos mintára. Pl. egy keresőmezőbe csak alfanumerikus karaktereket fogadunk el.
A SessionID ellopása ellen lehetőség van sütiket HTTPOnly-nak megjelölni, így a JavaScript nem fér hozzá, mivel erre általában nincs is szükség. 14-es verziótól felfele a Chrome figyeli, hogy ha futtatható kód van a kérésben, akkor letiltja a JavaScript futtatást, ez megnehezíti a reflektív támadást.

3.3. CSRF

Áttekintés

A Cross-Site Request Forgery (CSRF) során a támadó az áldozat nevében küld kéréseket a megtámadott oldalra. Ennek során a támadó paraméterezi fel a kérést, az áldozat küldi el, az oldal pedig végrehajtja. Ez a web állapotmentessége miatt tud működni, ugyanis nincs olyan információ, hogy a böngészőn belül melyik tabról lett megnyitva a weboldal, valamint a tab becsukásakor a felhasználó általában nem lesz kijelentkeztetve.

A támadás megvalósítása

Képzeljünk el egy üzenetküldő oldalt, ahol a bejelentkezett felhasználó egyetlen GET kéréssel tud üzenetet küldeni egy ismerősének. Amennyiben a támadó ismeri a pontos URL-t és a paramétereket (ezt általában ismertnek tekintjük), akkor össze tud állítani egy olyan oldalt, ahonnan a látogató böngészője elküld egy üzenetet a megtámadott oldalon.
Ehhez a támadó egy olyan oldalt állít össze, amelyben van egy ilyen tag:

```
<img src="http://localhost/sendmessage?to=bela&amp;msg=szia"/>
```

Amikor az áldozat megnyitja ezt az oldalt (figyeljük meg, hogy nem kell azonos domain alatt lennie, tehát a támadó ezt az oldalt minden esetben össze tudja állítani), akkor a böngésző megpróbálja letölteni a képet, ami egy GET kérés lesz. Ezzel együtt elküldi a SessionID-jét is, és ha történetesen be is van jelentkezve a támadott oldalra, akkor az üzenete el lesz küldve.

Védekezési lehetőségek

Ha GET helyett csak POST-tal lehet műveletet végezni, az nem teszi biztonságosabbá az alkalmazást, ugyanis POST kérést egyszerűen össze lehet rakni JavaScripttel. HTTP Referrer ellenőrzése ugyancsak nem jó, mert egyes böngészők ezt nem küldik el.
Általában a védekezés arra épül, hogy a form-ba kiküldünk a kliensnek egy véletlen számot, és figyeljük, hogy az megfelelően visszajön a form visszaérkezésekor. Mivel a támadónak alapvetően nincs lehetősége a form tartalmát megismerni (csak más támadással együtt, pl. az XSS használható erre), akkor a visszajövő véletlen nem fog egyezni, ezért elutasítjuk a kérést.
A védekezés az alábbi lépésekből áll:

- A form generálásakor készítünk egy véletlen számot, egy hidden fieldben megjelenítjük (`<input type="hidden" name="csrf" value="véletlenszám">`), valamint a session-ben eltároljuk

- A form feldolgozásakor kiolvassuk a session-ből az eltárolt értéket, és ha nem egyezik meg a visszakapottal, akkor hibával megszakítjuk a feldolgozást

Ennek megvan az a hátránya, hogy minden form generálása felülírja a sessionben elmentett értéket, emiatt a formokat ki kell töltenie a felhasználónak mielőtt másikat nyit meg. Ez több tabos böngészésnél kényelmetlen lehet. Erre félmegoldás, ha minden formnak van egyedi azonosítója, így csak ugyanazon űrlap megnyitásakor íródik felül a véletlen.

4. További megfontolások

Jelszavak tárolása adatbázisokban

Amennyiben a webes alkalmazásunk saját felhasználó-jelszó adatbázist használ, a jelszavakat tilos közvetlenül az adatbázisba illeszteni. Mindig használjunk egy egyirányú hash függvényt (pl. SHA-1 vagy SHA-256) a jelszón, illetve a hash-elés előtt a jelszó elé vagy után illesszünk egy minimum 10 karakter hosszúságú, felhasználónként különböző álvéletlen értéket, ún. saltot. Pszeudo-kóddal kifejezve:

```
stored_password := SHA-1(clear_test_password ||
  random_salt_different_for_every_user)
for (i=0; i<10000; i++) {
  stored_password := SHA-1(stored_password);
}
```

Ez a salt érték megakadályozza, hogy a hash értékekre ún. előszámított szivárvány-táblák segítségével percek alatt legyen feltörhető a jelszó. Az Oracle 11g-ben az adatbázis felhasználók jelszava három módon lehet letárolva: Vagy az új, 11g kompatibilis, SHA-1 alapú, a felhasználói nevet saltként használó jelszó hashként van letárolva, vagy a régi, saját hash algoritmust használó, nem case sensitive jelszó hash – szintén a felhasználói nevet saltként használva, vagy mindkettő hash. Érdemes megfigyelni, mivel nem véletlen a jelszóhoz tartozó salt, ezért a felhasználókra külön szivárvány-táblát lehet építeni, pl. a SYS v. SYSTEM adminisztátor felhasználó jelszó hash-eire az internetről letölthetőek ilyen szivárvány táblák.

Reguláris kifejezés alapú DoS támadás

ld. III. Függelék: Reguláris kifejezések (8)

5. Védekezési módszerek összefoglalása

A fentebb ismertetett támadások ellen tehát tartsuk szem előtt a következőket:

- az adatbázis-kezelőt futtassuk csökkentett jogosultságú felhasználó nevében
- az adatbázis-kapcsolathoz használt felhasználó csökkentett jogokkal rendelkezzen az adatbázis sémákban, ne legyen a séma tulajdonosa
- ne használjunk alapértelmezett, vagy egyszerű jelszavakat az adatbázisban, a nem szükséges felhasználókat tiltsuk le
- végezzünk szigorú input és output validációt
- kezeljük a speciális karaktereket escape-eléssel
- használjunk adatkötést avagy ún. paraméterezett SQL kódot (pl. oci_bind_by_name)
- ha nincs lehetőség paraméterezett SQL kódra, akkor használjunk tárolt eljárásokat
- a jelszavakat salttal egészítsük ki és hash-elve tároljuk el

6. Referencia, hasznos linkek, érdekességek

A webes támadások és az ismert védekezések legteljesebb gyűjtőhelye az OWASP (Open Web Application Security Project) www.owasp.org-on elérhető weblapja.

- Cross-Site Scripting
  https://www.owasp.org/index.php/XSS

- Cross-Site Request Forgery
  https://www.owasp.org/index.php/CSRF
- SQL injection cheat sheet
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Useful Oracle security tools
  http://www.petefinnigan.com/tools.htm
- Decrypt any password from password hash and successful network login packet capture (Oracle11g)
  http://www.soonerorlater.hu/index.khtml?article_id=512
- Red database security
  http://www.red-database-security.com/
- David Litchfield
  http://www.davidlitchfield.com/security.htm
- OWASP SQL Injection
  http://www.owasp.org/index.php/SQL_Injection
- NGS Software
  http://www.ngssoftware.com/media-room/WhitePapers.aspx
- XKCD
  http://xkcd.com/327/
- SQLMAP
  http://sqlmap.sourceforge.net/
- Szivárvány táblák működése
  http://en.wikipedia.org/wiki/Rainbow_table
- Advanced out-of-band SQL injection
  http://www.red-database-security.com/wp/confidence2009.pdf