# Adatbázisok és naptárak

Az iCalendar szabványt már megismertük, most itt az ideje, hogy az adatokat adatbázisból vegyük, és a naptárakat futásidőben állítsuk elő.

A múlt hónapban folytattuk az ismerkedést az *iCalendar* szabvánnyal, amelyre alapozva a programok naptár- és találkozóadatokat cserélhetnek egymással. Mint láttuk, egy *iCalendar* fájl egy vagy több eseményt és feladatot, tennivalót tárolhat. Ha a fájlt elérhetővé tesszük egy http kiszolgálón, például *Apache*-on keresztül, akkor bárki számára terjeszthetjük, aki az *iCalendar* kezelésére alkalmas programmal rendelkezik – ilyen például a *Mozilla Sunbird*. Amint a múlt hónapban láttuk, ennél eggyel tovább is léphetünk, és az *iCalendar* fájlt dinamikusan, *CGI* programmal is előállíthatjuk.

A múlt hónapban bemutatott programokat ugyan bizonyos körülmények között lehet hasznosítani, az minden webes fejlesztő számára egyértelmű, hogy a dátum- és időadatokat a programon belül tartani finoman szólva ostobaság volna.

Az ilyen adatok tárolását a legjobban relációs adatbázisban oldhatjuk meg, például *PostgreSQL* segítségével. Relációs adatbázis alkalmazásával könnyebb biztosítani a bevitt adatok érvényességét, továbbá gyors és rugalmas hozzáférést kapunk az elmentett adatok egy részéhez vagy egészéhez. További előny, hogy ha a naptáradatokat adatbázisban tároljuk, akkor ugyanazt a forrást használva egy-egy naptárfájlból több változatot is elő tudunk állítani.

Ebben a hónapban egy egyszerű, web alapú programra fogunk példát nézni, mely relációs adatbázisból veszi a naptáradatokat, ezek alapján előállít egy *iCalendar* adatfájlt, ami viszont alkalmas lesz az *iCalendar*-ral együttműködő programokba – mint a már említett *Mozilla Sunbird* – történő importálásra.

## A tábla megadása

Ha a naptáradatokat relációs adatbázisban kívánjuk tárolni, meg kell adnunk legalább egy táblát. Ennek oka az, hogy a relációs adatbázisok minden adatot – sokszor még a beállításokat és az állapotadatokat is – kétdimenziós táblázatokban tárolnak, amelyekben az oszlopok határozzák meg az egyes mezőket, míg egy-egy sor egy-egy rekordot tartalmaz. *PostgreSQL* alatt például a következő módon tudunk megadni egy egyszerű táblát:

```
CREATE TABLE Events (
  event_id        SERIAL    NOT NULL,
  event_summary   TEXT      NOT NULL
    CHECK (event_summary <> ''),
  event_location   TEXT      NOT NULL
    CHECK (event_location <> ''),
  event_start     TIMESTAMP NOT NULL,
  event_end       TIMESTAMP NOT NULL,
  event_timestamp TIMESTAMP NOT NULL
    DEFAULT NOW(),
  PRIMARY KEY(event_id)
);
```

A fenti tábla hat oszlopot tartalmaz. Az első az event_id (eseményazonosító), típusa `SERIAL`. Ha az `event_id`-nek nem adunk kifejezett értéket, amikor hozzáadunk egy-egy sort a táblához, akkor a *PostgreSQL* önműködően vesz egy legfeljebb $2^{31}$ értékű egész számot. A *PostgreSQL* nagyobb felső határ megadását vagy az 1-es értékhez való körbeforgást is lehetővé teszi, erről részletesebb tájékoztatás a leírásában található.

Az `event_id` oszlop tartalma egyedileg azonosítja a tábla sorait, ezt a `PRIMARY KEY` (elsődleges kulcs) kulcsszó (vagyis inkább kulcskifejezés) tudatja az adatbázissal. Ez nemcsak a rekordok lekérdezéséhez szükséges oszlopot adja meg a többi adatbázis-programozónak, de az értékek egyediségét és az oszlop indexelését is biztosítja.

Szintén automatikusan kap értéket az `event_timestamp` (esemény időbélyeg). A megadásból látható, arra is van mód, hogy explicit értéket adjunk neki (ezt meg is fogjuk tenni), de alapértelmezett értéke az aktuális idő. Az általános az, hogy ha ilyen formában adunk meg egy oszlopot, akkor explicit értéket csak kivételes alkalmakkor kap, és a legtöbb esetben a *PostgreSQL*-re bízzuk feltöltését az aktuális dátummal és idővel.

Megjegyezném, hogy az `event_summary` (esemény összegzése) és az `event_location` (esemény helye) egyaránt `TEXT` típusú (vagyis ezek végtelen hosszúságú, szöveges mezők), míg az `event_start` (esemény kezdete), az `event_end` (esemény vége) és az `event_timestamp` (esemény időbélyege) `TIMESTAMP` (időbélyeg) típusú; az *SQL*-ben ez a szabványos megoldás dátum és idő megadására.

*1. kódrészlet* db-calendar.py

```
#!/usr/bin/python
# A CGI modul beemelése
import cgi
import psycopg
from iCalendar import Calendar, Event
from datetime import datetime
from iCalendar import UTC # időzóna
# Az esetleges hibák naplózása
import cgitb
cgitb.enable(display=0, logdir="/tmp")
# content-type fejrész küldése
print "Content-type: text/calendar\n\n"
# Naptár objektum létrehozása
cal = Calendar()
# Milyen termék hozta létre a naptárat?
cal.add('prodid',
        '-//Python iCalendar 0.9.3//mxm.dk//')
# Az RFC 2445-nek a 2.0-s változat felel meg
cal.add('version', '2.0')
# Adatbázis-kapcsolat létesítése
db_connection =
  psycopg.connect('dbname=atf user=reuven')
db_cursor = db_connection.cursor()
db_cursor.execute
  ('''SELECT event_id, event_summary,
  ➥event_location,
            event_start, event_end,
            ➥event_timestamp
        FROM Events
    ORDER BY event_start''')
result_rows = db_cursor.fetchall()
for row in result_rows:
    # Esemény létrehozása
    event = Event()
    # Az eseményazonosító megadása
    event['uid'] = str(row[0]) + 'id@ATF'
    # Leírás és helyszín megadása
    event.add('summary', row[1])
    event.add('location', row[2])
    # A dátumok átalakítása
    event.add('dtstart', datetime(tzinfo=UTC(),
              *row[3].tuple()[0:5]))
    event.add('dtend', datetime(tzinfo=UTC(),
              *row[4].tuple()[0:5]))
    event.add('dtstamp', datetime(tzinfo=UTC(),
              *row[5].tuple()[0:5]))
    # Kapjon kiemelt fontosságot!
    event.add('priority', 5)
    # Az esemény hozzáadása a naptárhoz
    cal.add_component(event)
# A naptár utasítása önmaga leképezésére iCalendar
# fájlként, majd a fájl átadása a HTTP-válaszban.
print cal.as_string()
```

A tábla összes oszlopánál szerepel a `NOT NULL` (nem nulla) kifejezés, vagyis ezek nem kaphatják az ***SQL*** esetében a meghatározatlant jelentő `NULL` értéket. A `NULL` nem azonos az igazzal vagy a hamissal, ami a kezdők esetében sokszor félreértéseket okoz. A `NULL`-t mint ismeretlen, meg nem adott értéket kell kezelni – így talán könnyebb. Bár a `NULL` értékek segítségével könnyen meg tudjuk különböztetni a hamis és az ismeretlen értékeket, használatukat érdemes a lehető legszűkebb körre korlátozni. Sőt, általában azt szokták tanácsolni, hogy alapesetben minden oszlopot `NOT NULL`-ként, vagyis nem nullként adjunk meg, és csak akkor engedélyezzük a `NULL` értékek használatát, ha az valóban szükséges.

Felhívnám még a figyelmet a két szöveges oszlop megadására (`event_summary` és `event_location`), melyek egyrészt a `NOT NULL` kulcsszóval vannak ellátva, másrészt egy további, az üres karakterláncok bevitelét megakadályozó ellenőrzés tárgyát képezik. Az, hogy ezek az ellenőrzések helyénvalók-e, a tényleges alkalmazási környezettől függ. Előfordulhat, hogy valamiért inkább engedélyezni akarjuk a `NULL` értékek használatát, és az összegzésben vagy a hely megjelölésében szereplő üres karakterláncok sem okoznak gondot.

Bár a fenti táblamegadást csak egyszerű példának szántam, gondoljunk bele, milyen jó volna a helyszíneket külön táblában tárolni, például egy `location_id` (hely azonosítója) és `location_name` (hely neve) oszloppal, majd a szöveges `event_location` oszlopot a `location_id`-vel helyettesíteni. Így egységesíteni tudnánk a helyszínneveket, vagyis maga az adatbázis is egységesebbé válna, illetve arra is módunk nyílna, hogy kikeressük az adott helyszínen várható eseményeket.

Amikor végeztünk a tábla megadásával, hozzáadunk néhány indexet. Az indexek révén az adatok lekérdezése gyorsabb lesz a normál lekérdezéseknél; ennek az `INSERT` (beillesztési) műveletek megnövekedett végrehajtási idejével fizetjük meg az árát. Nézzük a megadásokat:

```
CREATE INDEX event_location_idx
  ON Events(event_location);
CREATE INDEX event_start_idx
  ON Events(event_start);
CREATE INDEX event_end_idx
  ON Events(event_end);
```

## Új adatok beillesztése

Elkészült a tábla és az indexek, kezdjük el eseményekkel feltölteni az adatbázist. Az események beillesztésére az `INSERT` parancs szolgál, melynek írásmódja a következő:

```
INSERT INTO Events
  (event_summary, event_location,
   event_start, event_end)
VALUES
  ('Március idusa', 'Mindenhol',
 '2005-March-15 00:00', '2005-March-15 23:59:59')
```

Mint látható, a fenti `INSERT` utasításban az `Events` tábla hat oszlopából csak négy szerepel. Ha megvizsgáljuk az új sor tartalmát, a következőt találjuk:

```
atf=# select * from events;
-[ RECORD 1 ]---+---------------------------
event_id        | 1
event_summary   | Március idusa
event_location  | Mindenhol
event_start     | 2005-03-15 00:00:00
event_end       | 2005-03-15 23:59:59
event_timestamp | 2005-04-04 01:20:15.575032
```

Mint látható, az `event_id` (mely, mint emlékszünk, `SERIAL` típusú) automatikusan kapott egy 1-es értéket, az `event_timestamp` pedig megkapta értékül a beillesztési kérés végrehajtásának időpontját.
Könnyű elképzelni, hogyan hívnánk meg az `INSERT` utasítást valamilyen web alapú programból, például *CGI*-n, esetleg valamilyen fejlettebb rendszeren keresztül, mint a *mod_perl* vagy a *Zope*. Hozzátenném, az adatbázisba befutó adatok beérkezésének módjával sem kell foglalkoznunk, különösen, ha megadtuk a megfelelő megszorításokat. Nyugodtan feltételezhetjük, hogy az adatbázisban található adatok megbízhatóak, és a kiszolgáló eleve elutasította az előírásoknak meg nem felelő beviteleket.

## Dinamikus iCalendar fájl létrehozása

Ha már van néhány eseményünk az adatbázistáblában, megpróbálhatjuk lekérdezni őket egy *CGI* programból. A program kimenete *iCalendar* formátumú lesz, vagyis az *iCalendar* ügyfelek is hozzáférhetnek majd az adatokhoz. Az *1. kódrészlet* egy ilyen programot tartalmaz, ez egyébként a múlt hónap *dynamic-calendar.py* programjának módosított változata. Mint a múltkor már utaltam rá, a programot nem kis részben azért *Pythonban* írtam, mert viszonylag szűkében állunk az *iCalendar* formátumú fájlok előállítására használható moduloknak. Szerencsére *Python* alá van ilyen modul, és én nem voltam rest kihasználni ennek előnyeit. Mint az 1. kódrészletből is látható, semmi bonyolultra nem kell számítani. Beemelünk néhány modult, létrehozunk egy naptár objektumot, majd beillesztjük az *iCalendar* szabvány által megkövetelt mezőket, megadva a naptár forrását. Ez után csatlakozunk egy a feltételezésünk szerint a helyi gépen futó *PostgreSQL* kiszolgálóhoz. Bár *Python-PostgreSQL* párosításhoz több adatbázis-csatoló is létezik, én a *psycopg*-t használom, mely gyors és üzembiztos. A *PostgreSQL*-hez a *psycopg* segítségével a következő írásmódot alkalmazva kapcsolódhatunk:

```
db_connection = psycopg.connect
                ('dbname=atf user=reuven')
```

A fentiek szerint az adatbázis neve `atf`, a felhasználónév pedig `reuven`. További lehetőség a kiszolgáló és valamilyen jelszó megadása, amire inkább akkor lehet szükség, ha éles rendszeren dolgozunk.
Miután csatlakoztunk az adatbázishoz, veszünk egy mutatót, ezzel tudjuk elküldeni a kéréseinket és lekérdezni az eredményüket:

```
db_cursor = db_connection.cursor()
```

Kezünkben a kurzorral immár tudunk *SQL*-lekérdezéseket küldeni az adatbázisnak; ezeket a *Python* „három idézőjeles" írásmódját kihasználva tettem könnyebben olvashatóvá. A következő lépés az eredmények lekérése. Ha nagyobb számú, akár több száz találatra számítanánk, akkor célszerűbb volna egyenként, esetleg csoportokban lekérni őket, ez esetben azonban tudjuk, hogy naptárunk mindössze néhány eseményt tartalmaz, ezért a `fetchall()` hívással egyetlen lépésben lekértem az összest.

```
result_rows = db_crsor.fetchall()
```

A `result_rows` (eredmény sorok) minden egyes eleme egyegy sor a *PostgreSQL* adatbázisból. Ha tehát egy `for` ciklussal végiglépkedünk a sorokon, akkor megkapjuk a találat egyes elemeit.
A legtöbb esetben az egész eljárás rutinműveletnek számít, ám amikor dátumokkal és időpontokkal dolgozunk, már több figyelmet kell szentelnünk az adatkezelésnek – márpedig a dátumok a naptár fontos elemei. A gond az, hogy a `psycopg` az *eGenix.com* nyílt forrású mxDateTime modulját használja, amellyel rendkívül könnyű a dátumokat kezelni. Csakhogy az *mxm iCalendar* modulja a Python datetime modulját használja, ami viszont eltérő. Le kell tehát kérdeznünk az egyes dátumokat (az egyes események kezdő és záró dátumát), *mxDateTime* példányból *datetime*-megfelelő tuple-é kell alakítanunk őket, majd a tuple felhasználásával elő kell állítanunk egy *datetime*-példányt, amelyet már át tudunk adni az `event.add`-nek:

```
event.add('dtstart', datetime(tzinfo=UTC(),
event.add('dtend', datetime(tzinfo=UTC(),
          *row[3].tuple()[0:5]))
```

A fenti három sornyi kódban a `datetime()` második átadott értéke pontosan az imént felvázolt eljárást hajtja végre. A kapott sorból vesz egy oszlopot, majd tuple-é alakítja. Ezután fogjuk a sorozat egy darabját (a *Python* kényelmesen használható [0:5] írásmódjával), ezzel hozzájutunk a `tuple()` által visszaadott elemek egy részhalmazához.
A `datetime()`-nak viszont nem lehet sorozatot átadni, csak különálló elemek sorozatát. Másként fogalmazva, a `datetime()` több számot vár, és nem hivatkozást vagy mutatót számok egy listájára. A tuple-t a *Python* * operátorával bontjuk önálló elemekre. Végül, az élesebb szemű olvasók bizonyára észrevették, hogy a `tzinfo` átadott érték még a tuple elemei előtt szerepel, ennek oka az, hogy *Pythonban* a nevesített átadott értékeket még a * operátor előtt kell megadni.

## További lehetőségek

A *db-calendar.py* futtatásával az *iCalendar* szabványnak tökéletesen megfelelő, a *Sunbird* vagy más naptárprogram alá gond nélkül importálható fájlt kapunk. Emellett, ha elvégzünk egy egyszerű módosítást az `Events` táblán, biztosíthatjuk, hogy a naptárra feliratkozó felhasználók mindig a legújabb változatot kapják meg.

Ennél tovább is mehetünk, érdemes például a *db-calendar.py*-t úgy módosítani, hogy kimenetében csak bizonyos események szerepeljenek. Lehetséges például, hogy megelégszünk a jövőbeli események átadásával, és nem terheljük mások naptárát (és sávszélességét) a múlt történéseivel. Ilyenkor elég egy egyszerű `WHERE` utasítást adnunk az *SQL*-lekérdezéshez, és a már megtörtént eseményeket könnyedén kiszűrhetjük.
Ennél érdekesebb lehetőség a különféle naptárbeli csoportok és elérési szintek elkülönítése. A *HTTP* támogatja a felhasználónév és jelszó alapú hitelesítést, és bár a *Sunbird* jelenleg nem teszi lehetővé az ilyen jellegű védelem alkalmazását, a jövőben várható ez irányú továbbfejlesztése (ahogy a többi programé is). Figyelembe véve, hogy a *CGI* programok könnyen meg tudják határozni a hitelesített *HTTP* kérésekhez tartozó felhasználónevet, talán nem túlzás azt mondani, hogy a *db-calendar.py* a különböző felhasználók számára különböző kimeneteket is elő tudna állítani, a hozzárendelt engedélyek és szerepek alapján.
Végül utalnék rá, hogy bár az elmúlt hónapokban az *iCalendar* formátumra összpontosítottunk, nincs semmi oka annak, hogy az adatbázis tartalmát kizárólag *iCalendar* fájlként tegyük elérhetővé. Miért ne tehetnénk meg például, hogy az *iCalendar* mellett a jó öreg *HTML* formátumban is megjelenítjük az eseményeket? *HTML* táblázatok segítségével roppant egyszerű volna a feladat megoldása – ez is kiváló példája annak, hogy relációs adatbázisra támaszkodva az adatok különféle formátumokban való megjelenítése nem probléma, sokkal inkább lehetőség.

## Összefoglalás

Röviden áttekintettük, hogyan alkalmazhatunk adatbázist az események adatainak tárolására, amelyek alapján végül *iCalendar*-megfelelő fájlt állítottunk elő. Adatbázis használatával nemcsak a tárolt adatok megbízhatóságát növelhetjük, de könnyen és gyorsan állíthatunk elő dinamikus, az *iCalendar* formátum olvasására képes programok által kezelhető fájlokat.

*Linux Journal 2005. július, 135. szám*

***Reuven M. Lerner*** (➲ http://www.lerner.co.il/atf) Nyílt forrású programokra, valamint web- és adatbázis-alkalmazásokra szakosodott tanácsadó. Könyve, a Core Perl, 2002 januárjában jelent meg a Prentice Hall gondozásában. Reuven feleségével és lányaival nemrég költözött Chicagóba.

### KAPCSOLÓDÓ CÍMEK

- ➲ www.postgresql.org
- ➲ www.python.org
- ➲ www.mxm.dk/products/public/ical/downloads
- ➲ techdocs.postgresql.org/techdocs/faqdatesintervals.php
- ➲ initd.org/projects/psycopg1
- ➲ www.mozilla.org/projects/calendar/download.html
- ➲ www.ietf.org/rfc/rfc2445.txt