

plphp - PHP jako język proceduralny w PostgreSQL

triggery

W poprzednim tekście o pPHP przedstawione zostały podstawowe informacje dotyczące nowego zastosowania PHP. PHP jako język proceduralny daje olbrzymie możliwości rozbudowy funkcjonalności baz danych w PostgreSQL. Jak każdy język proceduralny pPHP można także wykorzystać do pisania własnych triggerów, czyli funkcji wywoływanych przez PostgreSQL przy zajściu określonego zdarzenia.

Triggery w PostgreSQL

Trigger jest powiązaniem stworzonej przez nas funkcji, tabeli oraz akcji, która go wywołuje. W momencie wywołania INSERT, UPDATE lub DELETE trigger może wywołać na tabeli napisaną wcześniej funkcję.

Dzięki triggerom można wpływać na wykonywane akcje, doprowadzać do ich anulowania, zmieniać w locie dane lub wykonać zupełnie dodatkowe zadania zupełnie niezwiązane z akcją, która je wywołała. W ten sposób dużą część pracy i nadzorowania nad bazą danych i danymi w niej przechowywanymi można powierzyć jej samej, co znacznie upraszcza proces budowania aplikacji i ogranicza ilość błędów możliwych do popełnienia.

Najważniejsze informacje o triggerach

Przy tworzeniu nowych triggerów warto pamiętać o ich podstawowych cechach:

- mogą być wywoływane jedynie przez INSERT, UPDATE lub DELETE
- wywołanie triggerów może nastąpić dla każdego zmienianego rzędu lub całego zdania SQL
- funkcja obsługująca trigger musi być utworzona zanim stworzony zostanie sam trigger
- jedna funkcja może być wykorzystywana przez różne triggery
- zwracając odpowiednią wartość w funkcji obsługującej trigger można spowodować, że operacja go wywołująca zostanie pominięta
- jeśli na jednej tabeli zdefiniowanych jest wiele triggerów, wykonywane są one w porządku alfabetycznym

Powyzsze informacje są podstawowymi informacjami dotyczącymi triggerów, które należy mieć na uwadze podczas ich tworzenia. Dokładny opis tworzenia triggerów znajduje się w rozdziale 35. dokumentacji PostgreSQL "Triggers".

Informacje o triggerach w systemie przechowywane są w tabeli pg_trigger. Tam znajdują się powiązania funkcji wywoływanych oraz tabel.

```
plphp_db=# \d pg_trigger
Table "pg_catalog.pg_trigger"
Column          |      Type      | Modifiers
-----+-----+-----
tgrelid         |      oid       | not null
tgname          |      name      | not null
tgfoid         |      oid       | not null
```

tgtype	smallint	not null
tgenabled	boolean	not null
tgisconstraint	boolean	not null
tgconstrname	name	not null
tgconstrrelid	oid	not null
tgdeferrable	boolean	not null
tginitdeferred	boolean	not null
tgargs	smallint	not null
tgattr	int2vector	not null
tgargs	bytea	

Indexes:

```
"pg_trigger_oid_index" unique, btree (oid)
"pg_trigger_tgrelid_tgname_index" unique, btree (tgrelid, tgname)
"pg_trigger_tgconstrname_index" btree (tgconstrname)
"pg_trigger_tgconstrrelid_index" btree (tgconstrrelid)
```

Po świeżej instalacji PostgreSQL utworzone są jedynie dwa triggery pomagające zarządzanie użytkownikami.

```
plphp_Db=# SELECT pt.tgname, pt.tgenabled, pc.relname, pp.proname
          FROM pg_trigger pt, pg_class pc, pg_proc pp
          WHERE pt.tgrelid=pc.oid and tgfoid=pp.oid ;
```

tgname	tgenabled	relname	proname
pg_sync_pg_pwd	t	pg_shadow	update_pg_pwd_and_pg_group
pg_sync_pg_group	t	pg_group	update_pg_pwd_and_pg_group

(2 rows)

W pierwszej kolumnie znajduje się podana nazwa triggera z tabeli pg_trigger, w następnej jego stan, tutaj – włączony. Trzecia kolumna to nazwa tabeli, z którą został powiązany, i na końcu nazwa procedury, która będzie wywołana przy wywołaniu triggera.

Przykłady zastosowania

Poniżej przedstawione zostanie kilka przykładów pokazujących możliwości jakie dają nam triggery. Oczywiście jest to tylko drobny wstęp do tego co można uzyskać dzięki plphp i triggerom.

Przypadek, którym się zajmiemy będzie bardzo prosty: trzy tabele, pierwsza opisująca pracowników, druga samochody i ostatnia łącząca dwie poprzednie.

```
-- Tworzymy tabelę pracownicy.
create table pracownicy (
    id          serial,
    imie        text,
    nazwisko    text,
    wynagrodzenie float
);

-- Tworzymy tabelę samochody.
create table samochody (
    id          serial,
    marka       text,
    numer_silnika text
);

-- Tworzymy tabelę samochody_pracownikow.
```

```

create table samochody_pracownikow (
    id_pracownika int,
    id_samochodu int
);

```

Tabela `samochody_pracownikow` wiąże pracowników z samochodami, które są dostępne w firmie. Dobrze byłoby żeby poza logami aplikacji, która powstanie na bazie tych tabel, generowane były dodatkowe informacje, które będą pomocne w ustaleniu pomyłek i błędów.

W tym celu w tabeli `samochody_pracownikow_log` zapisywana będzie każda zmiana jaka dokona się na tabeli `samochody_pracownikow`.

```

-- Tworzymy tabelę samochody_pracownikow_log.
create table samochody_pracownikow_log (
    id serial,
    data timestamp default now(),
    zmiana text,
    id_pracownika int,
    id_samochodu int
);

```

Tabela `samochody_pracownikow_log` poza polami identycznymi z tabelą `samochody_pracownikow` zawiera dwa dodatkowe pola: `data`, która określa czas zdarzenia, oraz pole `zmiana`, które będzie zawierało jedną z trzech wartości: `INSERT`, `UPDATE`, `DELETE`, w zależności od wykonanej operacji na tabeli.

```

-- Funkcja wykorzystywana przy triggerach musi zwracać wartość "trigger".
CREATE OR replace function splf() RETURNS trigger AS '
    global $_TD;

    /* Jeśli dane są kasowane, przechowujemy dane skasowane. */
    if ($_TD["event"]="DELETE")
    {
        spi_exec_query("INSERT
            INTO samochody_pracownikow_log (zmiana, id_pracownika, id_samochodu)
            VALUES ('".$_TD["event"]."',".$_TD["old"]["id_pracownika"]."',".$_TD["old"]["id_samochodu"]."');");
    }
    /* Jeśli dane są wprowadzane (INSERT) lub uaktualniane (UPDATE), wprowadzamy
    nowe dane do tabeli z logami. */
    else
        spi_exec_query("INSERT
            INTO samochody_pracownikow_log (zmiana,id_pracownika, id_samochodu)
            VALUES ('".$_TD["event"]."',".$_TD["new"]["id_pracownika"]."',".$_TD["new"]["id_samochodu"]."');");

    return;
' LANGUAGE 'plphp';

```

Funkcja ta ma zadeklarowaną zmienną globalną `$_TD`, która wykorzystywana jest do przenoszenia informacji o triggerach. W tabeli tej znajduje się wiele informacji, które mogą zostać użyte w funkcjach wywoływanych przez trigger.

\$_TD["new"]	Tablica asocjacyjna zawierająca nowe wartości w przypadku kiedy są one wstawiane/aktualizowane przez INSERT lub UPDATE. W przypadku DELETE tablica ta jest pusta. Pola indeksowane są nazwami pól z tabeli. Pola, które zawierają wartość NULL nie są umieszczane w tej tablicy.
\$_TD["old"]	Tablica asocjacyjna zawierająca stare wartości w przypadku kiedy są one wstawiane/aktualizowane przez DELETE lub UPDATE. W przypadku INSERT tablica ta jest pusta. Pola, które zawierają wartość NULL nie są umieszczane w tej tablicy.
\$_TD["name"]	Zawiera nazwę triggera.
\$_TD["event"]	Zawiera jedną z wartości: INSERT, UPDATE, DELETE, lub UNKNOWN.
\$_TD["when"]	Zawiera jedną z wartości: BEFORE, AFTER, lub UNKNOWN.
\$_TD["level"]	Zawiera jedną z wartości: ROW, STATEMENT, lub UNKNOWN.
\$_TD["relid"]	Zawiera ID (lub raczej OID) tabeli, na której trigger został wywołany.
\$_TD["relname"]	Zawiera nazwę tabeli.
\$_TD["argc"]	Zawiera liczbę argumentów triggera.
\$_TD["args"]	Jeśli trigger został wywołany z argumentami, ich wartości znajdują się w zmiennych od \$_TD["args"][0] do \$_TD["args"][\$_TD["argc"]-1].

Funkcja wykorzystuje także możliwość wywoływania zapytań SQL. Za pomocą funkcji `spi_exec_query` wprowadzamy informację o dokonywanych zmianach na tabeli `samochody_pracownikow` do tabeli `samochody_pracownikow_log`.

Na koniec pozostało połączyć trigger z tabelą i akcjami, na których ma być wywoływany. Ustawiamy trigger tak żeby był wywoływany po dokonaniu zmiany, po to aby mieć pewność, że zmiany które logujemy na pewno zaszły.

```
CREATE TRIGGER splft AFTER INSERT OR UPDATE OR DELETE ON samochody_pracownikow
FOR EACH ROW EXECUTE PROCEDURE splf();
```

Zdanie to powoduje, że po (AFTER) każdym wywołaniu INSERT, UPDATE lub DELETE na rzędzie danych w tabeli wykonana zostanie funkcja `splf()`.

Sprawdźmy jak to działa.

W tabelach znajdują się następujące dane:

```
plphp_db=# select * from samochody;
 id | marka | numer_silnika
-----+-----+-----
  1 | VW Garbus | 111-222-444
  2 | Polonez Sporting | 333-111-777
(2 rows)
```

```
plphp_db=# select * from pracownicy;
 id | imie | nazwisko | wynagrodzenie
```

```

-----+-----+-----+-----
 2 | Jan      | Kowalski |      3000
 3 | Stanislaw | Pastewny |      2000
(2 rows)

```

Wiążemy pracownika o id=1 z samochodem id=2.

```

plphp_db=# insert into samochody_pracownikow values (1,2);
INSERT 58389 1

```

```

plphp_db=# select * from samochody_pracownikow;
 id_pracownika | id_samochodu
-----+-----
                1 |                2
(1 row)

```

W tabeli z logami trigger zapisał dodatkowe informacje, więc wszystko działa.

```

plphp_db=# select * from samochody_pracownikow_log;
 id | data | zmiana | id_pracownika | id_samochodu
-----+-----+-----+-----+-----
  1 | 2004-09-08 12:31:56.135713 | INSERT |                1 |                2
(1 row)

```

Jeśli mamy już dodatkowe logowanie, dobrze byłoby wiedzieć ilu jest pracowników i jakie są średnie zarobki w naszym przedsiębiorstwie. Jeśli pracowników jest kilkunastu lub kilkudziesięciu, to można łatwo to zrobić prostym zapytaniem SQLowym. Przy założeniu, że huta przy naszej działalności to małe piwo, statystyki będą przechowywane w osobnej tabeli, aby nie marnować czasu na ich obliczenia. Tabela ta będzie uaktualniana automatycznie, tak żeby nie trzeba było o niej pamiętać przy dokonywaniu zmian w stanie naszych pracowników.

```

-- Tworzymy tabelę statystyki przechowując interesujące nas dane dotyczące
-- stanu zatrudnienia i aktualnych średnich zarobków.
CREATE TABLE statystyki
(
    liczba_pracownikow      INT,
    srednie_zarobki         FLOAT
);

```

Tabela statystyki zawiera tylko dwa pola: liczba_pracownikow i srednie_zarobki – nic dodać, nic ująć.

Funkcja, która zostanie powiązana z tą tabelą ma za zadanie aktualizować liczbę pracowników oraz wartość średnich zarobków, w zależności od wykonywanej akcji. Warto zwrócić tutaj uwagę, że przy funkcjach spi_exec_query używane są wyniki zwracane przez SELECT wykonywany na tabeli pracownicy.

```

CREATE OR REPLACE FUNCTION pf() RETURNS trigger AS '
global $_TD;

$lp = 0;
$srednie_zarobki = 0;

```

```

/* Pobieramy aktualny stan statystyk. */
$wyn = spi_exec_query("SELECT * FROM statystyki;");
$lp = $wyn[0]['liczba_pracownikow'];
$sz = $wyn[0]['srednie_zarobki'];

/* Dodawany jest nowy pracownik. */
if ($_TD["event"]="INSERT")
{
    $sz = (($sz * $lp + $_TD["new"]["wynagrodzenie"]) / ($lp+1));
    $lp++;
    spi_exec_query("UPDATE statystyki set liczba_pracownikow=$lp;");
    spi_exec_query("UPDATE statystyki set srednie_zarobki=$sz;");
}
/* Redukcje... redukcje... redukcje... */
else if ($_TD["event"]="DELETE")
{
    $lp--;
    spi_exec_query("UPDATE statystyki set liczba_pracownikow=$lp;");
    $wyn=spi_exec_query("SELECT AVG(wynagrodzenie) as srednia
                        FROM pracownicy;");
    $sz=$wyn[0]['srednia'];
    spi_exec_query("UPDATE statystyki set srednie_zarobki=$sz;");
}
/* Ktoś dostał podwyżkę! */
else
{
    $wyn=spi_exec_query("SELECT AVG(wynagrodzenie) as srednia
                        FROM pracownicy;");
    $sz=$wyn[0]['srednia'];
    spi_exec_query("UPDATE statystyki set srednie_zarobki=$sz;");
}

return;
' LANGUAGE 'plphp';

CREATE TRIGGER pt AFTER INSERT OR UPDATE OR DELETE ON pracownicy
FOR EACH ROW EXECUTE PROCEDURE pf();

```

W tym miejscu warto zwrócić uwagę na jeszcze jedną rzecz. Triggery, jak widać powyżej, tworzone są ze słowem kluczowym BEFORE lub AFTER. Ma to bezpośredni wpływ na to jakie dane znajdą się w tablicy \$_TD. W przypadku wywołania przez BEFORE zmiany dokonywane przez INSERT, UPDATE, DELETE nie są widzialne wewnątrz triggera. Jeśli jednak trigger został wywołany podczas zmiany wielu rzędów danej tabeli, zmiany poprzednich rzędów będą widoczne. Jeśli trigger został wywołany z AFTER, wszystkie zmiany wykonane przez zewnętrzne polecenia SQL będą już widoczne.

Dodatkową rzeczą wykonywaną za pomocą triggera na tabeli statystyki jest zapobieganie dokładaniu dodatkowych pól lub usunięciu tego właściwego ze statystykami. Tabela ta z założenia ma posiadać jedynie jeden wpis.

```

CREATE OR REPLACE FUNCTION sf() RETURNS trigger AS '
    return "SKIP";
return;
' LANGUAGE 'plphp';

CREATE TRIGGER st BEFORE INSERT OR DELETE ON statystyki
FOR EACH ROW EXECUTE PROCEDURE sf();

```

Zwrócenie wartości SKIP przy wykonywanym INSERT lub DELETE

powoduje, że ich wykonanie jest przerywane czyli, że nie można nawet przez pomyłkę usunąć statystyk lub dodać niepotrzebne dane. Oczywiście należy pamiętać, aby zaraz po utworzeniu statystyk, a jeszcze przed utworzeniem tego triggera, wprowadzić do tej tabeli jeden rząd, w którym przechowywane będą właściwe statystyki. Trigger ten musi być także utworzony z opcją BEFORE, aby zdążył zablokować utworzenie/usunięcie danych z tabeli statystyki. Sam trigger został utworzony tak aby był wywoływany jedynie przy wywołaniu INSERT i DELETE, więc w samej treści funkcji nie ma potrzeby sprawdzania jaka akcja ją wywołała.

Sprawdźmy jak to działa.

Zatrudniamy dwóch nowych pracowników.

```
INSERT INTO pracownicy (imie,nazwisko, wynagrodzenie)
VALUES ('Jan','Kowalski',3000);
INSERT INTO pracownicy (imie,nazwisko, wynagrodzenie)
VALUES ('Stanislaw','Pastewny',2000);
```

```
plphp_db=# select * from statystyki;
 liczba_pracownikow | srednie_zarobki
-----+-----
                2 |                2500
(1 row)
```

Pracownik o id=1 dostaje podwyżkę do 5000.

```
plphp_db=# UPDATE pracownicy set wynagrodzenie=5000 where id=1;
UPDATE 1
```

```
plphp_db=# select * from statystyki;
 liczba_pracownikow | srednie_zarobki
-----+-----
                2 |                3500
(1 row)
```

Widać, że dane w tabeli statystyki zmieniają się zupełnie automatycznie, bez naszego udziału. Czyż to nie jest wygodne?:)

Spróbujmy jeszcze na koniec popsuć trochę tabelę statystyki przez dodanie nadmiarowego rzędu z danymi lub skasowanie wszystkich istniejących.

```
plphp_db=# insert into statystyki (liczba_pracownikow , srednie_zarobki) values
(1,1);
INSERT 0 0
```

```
plphp_db=# delete from statystyki;
DELETE 0
```

```
plphp_db=# select * from statystyki;
 liczba_pracownikow | srednie_zarobki
-----+-----
                2 |                2500
(1 row)
```

Triggery jak widać zapobiegły tragedii i dane statystyczne zostały ocalone.

Koniec...

Jak widać na tych prostych przykładach, korzystanie z triggerów może w znacznym stopniu uprościć budowanie aplikacji opartych o bazy danych. Właściwe wykorzystanie własnych funkcji może także znakomicie zwiększyć wydajność całej aplikacji, choćby przez posługiwanie się, podobnie jak tu, tabelami pomocniczymi. Oczywiście najlepszym rozwiązaniem byłoby przepisanie stworzonych przez nas triggerów na C, co znacznie zwiększyłoby wydajność, ale na etapie rozwoju aplikacji plphp wydaje się być znakomitą pomocą, a przy mniejszych projektach wystarczającym środkiem pozwalającym na bezpieczne dotarcie do celu.

Linki:

<http://plphp.commandprompt.com/> - oficjalne repozytorium plphp

<http://www.postgresql.org/docs/7.4/static/triggers.html> – triggery w

PostgreSQL