Session 15 – Functions & Trigger

> **Session-Typ:** Vorlesung **Dauer:** 90 Minuten **Lernziele:** Stored Functions schreiben, Trigger erstellen, Automatisierung verstehen

Willkommen zu Session 15! Heute schauen wir uns an, wie wir Logik nicht nur in unserer Anwendung, sondern direkt in der Datenbank ausführen können. Warum ist das sinnvoll? Stellen Sie sich vor, Sie möchten, dass bei jeder Änderung an einem Produkt automatisch ein Timestamp aktualisiert wird – oder dass jede Preisänderung protokolliert wird. Das manuell in jeder Anwendung zu implementieren ist fehleranfällig. Besser: Die Datenbank macht es automatisch! Heute lernen Sie Functions und Trigger kennen – und probieren alles direkt im Browser aus.

Motivation: Warum Logik in der Datenbank?

Beginnen wir mit einer Frage: Wo sollte Geschäftslogik leben? In der Anwendung oder in der Datenbank? Die Antwort ist: Es kommt darauf an! Aber für bestimmte Aufgaben ist die Datenbank der perfekte Ort.

Problem 1: Vergessene Timestamps

Klassisches Szenario: Sie wollen bei jeder Änderung an einem Datensatz das „updated_at“ Feld aktualisieren.

Ohne Automatisierung (Anwendungsseite):

```
// In jeder Update-Funktion manuell:
await db.query(
  'UPDATE products SET price = $1, updated_at = NOW() WHERE id = $2',
  [newPrice, productId]
);

// ❌ Fehleranfällig: Was, wenn jemand vergisst, updated_at zu setzen?
// ❌ Duplizierter Code: In 50 verschiedenen Update-Funktionen
// ❌ Inkonsistent: Manche Entwickler machen es, andere nicht
```

Mit einem Trigger ist das Problem gelöst – einmal definiert, funktioniert es immer. Automatisch. Konsistent. Ohne dass die Anwendung daran denken muss.

Mit Trigger (Datenbank):

```
CREATE TRIGGER set_updated_at
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();

-- ✅ Funktioniert immer, egal welche Anwendung zugreift
-- ✅ Code an einer zentralen Stelle
```

```
-- ✅ Konsistent für alle Updates
```

Problem 2: Audit-Logging

Zweites Szenario: Sie wollen nachvollziehen, wer wann welche Preise geändert hat. Compliance-Anforderung!

Ohne Trigger:

```
// In jedem Update manuell protokollieren
await db.query('UPDATE products SET price = $1 WHERE id = $2', [newPrice, i
await db.query(
  'INSERT INTO audit_log (table_name, action, old_value, new_value) VALUES
    $2, $3, $4)',
  ['products', 'UPDATE', oldPrice, newPrice]
);

// ❌ Zwei Queries - was bei Fehler zwischen beiden?
// ❌ Entwickler muss daran denken
// ❌ Audit-Log kann vergessen werden
```

Mit einem Trigger passiert das Logging automatisch – transparent, konsistent, fehlerfrei.

Mit Trigger:

```
CREATE TRIGGER audit_changes
AFTER UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION log_change();

-- ✅ Automatisch bei jedem Update
-- ✅ Kann nicht vergessen werden
-- ✅ Atomar: Entweder beide Operationen oder keine
```

Use Cases für Functions & Trigger

Wann machen Functions und Trigger Sinn? Hier ist eine Übersicht:

Use Case	Functions	Trigger
Berechnungen (z.B. Steuer, Rabatt)	✅ Wiederverwendbar	⚠️ Automatisch bei jedem Event
Validierung (z.B. negative Preise verhindern)	✅ Kann manuell aufgerufen werden	✅✅ Automatisch, kann nicht umgangen werden
Automatische Timestamps	⚠️ Muss aufgerufen werden	✅✅ Automatisch bei INSERT/UPDATE
Audit-Logging	⚠️ Muss explizit aufgerufen werden	✅✅ Automatisch, konsistent
Soft Delete (Löschen = Markieren)	⚠️ Muss implementiert werden	✅✅ Überschreibt DELETE automatisch
Komplexe Geschäftslogik	✅ Gut testbar, wiederverwendbar	⚠️ Schwer zu debuggen

Faustregel:

- **Functions** = Wiederverwendbare Logik, die Sie aktiv aufrufen
- **Trigger** = Automatische Reaktion auf Datenbankänderungen

Heute lernen Sie beide Konzepte kennen – und zwar nicht nur theoretisch, sondern mit vielen praktischen Demos, die Sie direkt im Browser ausprobieren können!

Teil 1: Stored Functions

Beginnen wir mit Stored Functions. Das sind quasi JavaScript-Funktionen, aber in der Datenbank. Sie schreiben sie einmal, speichern sie in der Datenbank – und können sie dann in Queries verwenden.

Was sind Stored Functions?

Eine Stored Function ist ein Stück SQL-Code, das in der Datenbank gespeichert wird und wiederverwendet werden kann.

Vorteile:

- ✅ **Wiederverwendbarkeit:** Einmal schreiben, überall nutzen
- ✅ **Performance:** Code läuft auf dem Datenbankserver (kein Netzwerk-Overhead)
- ✅ **Konsistenz:** Eine zentrale Definition, keine Duplikation
- ✅ **Sicherheit:** Benutzer können Funktionen aufrufen, ohne Tabellenzugriff zu haben

Nachteile:

- ⚠️ **Portabilität:** Syntax unterscheidet sich zwischen Datenbanken
- ⚠️ **Debugging:** Schwieriger als Anwendungscode
- ⚠️ **Testing:** Unit-Tests sind komplizierter

Grundlegende Syntax

Die Syntax für `CREATE FUNCTION` sieht in PostgreSQL so aus:

```
CREATE FUNCTION function_name(parameter1 TYPE, parameter2 TYPE, ...)
RETURNS return_type AS $$
BEGIN
    -- Funktionskörper
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Wichtige Bestandteile:

- `CREATE FUNCTION function_name(...)` – Name und Parameter
- `RETURNS return_type` – Was gibt die Funktion zurück? (INT, TEXT, DECIMAL, …)
- `$$ ... $$` – String-Delimiter (statt `'...'`), macht Code lesbarer
- `BEGIN ... END;` – Der eigentliche Code
- `LANGUAGE plpgsql` – PostgreSQL's Procedural Language

Demo 1: Einfache Addition

Schauen wir uns ein ganz einfaches Beispiel an: Eine Funktion, die zwei Zahlen addiert.

```
1  CREATE FUNCTION add_numbers(a INT, b INT)
2  RETURNS INT AS $$
3  BEGIN
4      RETURN a + b;
5  END;
6  $$ LANGUAGE plpgsql;
7
8  -- Aufruf
9  SELECT add_numbers(5, 3) as result;
```

```
CREATE FUNCTION add_numbers(a INT, b INT)
RETURNS INT AS $$
BEGIN
   RETURN a + b;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Aufruf
SELECT add_numbers(5, 3) as result
```

#	result
1	8

1 rows

Das war's! Sie sehen: Parameter in Klammern, Rückgabetyp mit RETURNS, und im Body ein einfaches RETURN. Probieren Sie es aus – ändern Sie die Zahlen!

Demo 2: String-Verarbeitung

Functions können auch mit Strings arbeiten. Hier eine Grußfunktion:

```
CREATE FUNCTION greet(name TEXT)
RETURNS TEXT AS $$
BEGIN
    IF name IS NULL THEN
        RETURN 'Hallo Unbekannter!';
    ELSE
        RETURN 'Hallo ' || name || '!';
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Aufruf
SELECT greet('Alice') as greeting;
SELECT greet('Bob') as greeting;
SELECT greet(NULL) as greeting;
```

```
CREATE FUNCTION greet(name TEXT)
RETURNS TEXT AS $$
BEGIN
   IF name IS NULL THEN
      RETURN 'Hallo Unbekannter!';
   ELSE
      RETURN 'Hallo ' || name || '!';
   END IF;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Aufruf
SELECT greet('Alice') as greeting
```

#	greeting
1	Hallo Alice!

1 rows

```
SELECT greet('Bob') as greeting
```

#	greeting
1	Hallo Bob!

1 rows

```
SELECT greet(NULL) as greeting
```

#	greeting
1	Hallo Unbekannter!

1 rows

Hier sehen Sie das erste Mal IF...THEN...ELSE. Schauen wir uns Kontrollstrukturen genauer an.

Kontrollstrukturen: IF & CASE

IF / THEN / ELSE

Mit IF können Sie Bedingungen prüfen – wie in jeder Programmiersprache.

Syntax:

```
IF condition THEN
    -- Code, wenn wahr
```

```
ELSE
    -- Code, wenn falsch
END IF;
```

Wichtig:

- `THEN` nach der Bedingung
- `END IF;` zum Abschließen (nicht nur `END`)

Demo 3: Alterscheck

Ein praktisches Beispiel: Prüfen, ob jemand volljährig ist.

```
CREATE FUNCTION check_age(age INT)
RETURNS TEXT AS $$
BEGIN
    IF age >= 18 THEN
        RETURN 'Volljährig';
    ELSE
        RETURN 'Minderjährig';
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Testen
SELECT check_age(25) as status;
SELECT check_age(16) as status;
SELECT check_age(18) as status;  -- Grenzfall
SELECT check_age(NULL) as status; -- Was passiert hier?
```

```
CREATE FUNCTION check_age(age INT)
RETURNS TEXT AS $$
BEGIN
    IF age >= 18 THEN
        RETURN 'Volljährig';
    ELSE
        RETURN 'Minderjährig';
    END IF;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Testen
SELECT check_age(25) as status
```

#	status
1	Volljährig

1 rows

```
SELECT check_age(16) as status
```

#	status
1	Minderjährig

1 rows

```
SELECT check_age(18) as status
```

#	status
1	Volljährig

1 rows

```
-- Grenzfall
SELECT check_age(NULL) as status
```

#	status
1	Minderjährig

1 rows

```
-- Was passiert hier?
```

ok

Beachten Sie: Bei NULL gibt die Funktion auch NULL zurück – denn NULL >= 18 ist NULL, also falsch. Das ist SQL-Logik!

CASE: Alternative zu IF

Für Mehrfachauswahl ist CASE oft eleganter als verschachtelte IFs.

Syntax:

```
RETURN CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default_result
END;
```

Demo 4: Notensystem

Ein Notensystem – perfekt für CASE:

```
1  CREATE FUNCTION get_grade(score INT)
2  RETURNS TEXT AS $$
3  BEGIN
4      RETURN CASE
5          WHEN score >= 90 THEN 'Sehr gut (1)'
6          WHEN score >= 80 THEN 'Gut (2)'
7          WHEN score >= 70 THEN 'Befriedigend (3)'
8          WHEN score >= 60 THEN 'Ausreichend (4)'
9          ELSE 'Nicht bestanden (5)'
10     END;
11 END;
12 $$ LANGUAGE plpgsql;
13
14 -- Testen
15 SELECT get_grade(95) as note;
16 SELECT get_grade(85) as note;
17 SELECT get_grade(72) as note;
18 SELECT get_grade(50) as note;
```

```
CREATE FUNCTION get_grade(score INT)
RETURNS TEXT AS $$
BEGIN
   RETURN CASE
      WHEN score >= 90 THEN 'Sehr gut (1)'
      WHEN score >= 80 THEN 'Gut (2)'
      WHEN score >= 70 THEN 'Befriedigend (3)'
      WHEN score >= 60 THEN 'Ausreichend (4)'
      ELSE 'Nicht bestanden (5)'
   END;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Testen
SELECT get_grade(95) as note
```

#	note
1	Sehr gut (1)

1 rows

```
SELECT get_grade(85) as note
```

#	note
1	Gut (2)

1 rows

```
SELECT get_grade(72) as note
```

#	note
1	Befriedigend (3)

1 rows

```
SELECT get_grade(50) as note
```

#	note
1	Nicht bestanden (5)

1 rows

CASE ist hier viel lesbarer als verschachtelte IFs. Wann nutzen Sie was? IF für komplexe Bedingungen mit mehreren Anweisungen, CASE für einfache Wertauswahl.

Fehlerbehandlung: RAISE

Was, wenn etwas schiefgeht? Mit RAISE können Sie Fehler werfen – ähnlich wie „throw“ in JavaScript.

RAISE EXCEPTION

Syntax:

```
RAISE EXCEPTION 'Fehlermeldung: %', variable;
```

Platzhalter: - `%` wird durch die nächste Variable ersetzt - Ähnlich wie `printf` in C oder String-Interpolation

Demo 5: Division mit Fehlerbehandlung

Ein Klassiker: Division durch Null verhindern.

```
CREATE FUNCTION divide(a INT, b INT)
RETURNS DECIMAL AS $$
BEGIN
    IF b = 0 THEN
        RAISE EXCEPTION 'Division durch Null ist nicht erlaubt! (Divi
          )', b;
    END IF;
    RETURN a::DECIMAL / b;
END;
$$ LANGUAGE plpgsql;

-- Testen: Erfolg
SELECT divide(10, 2) as result;
SELECT divide(100, 4) as result;

-- Testen: Fehler
SELECT divide(10, 0) as result;  -- ❌ Wirft Exception
```

```
CREATE FUNCTION divide(a INT, b INT)
RETURNS DECIMAL AS $$
BEGIN
   IF b = 0 THEN
      RAISE EXCEPTION 'Division durch Null ist nicht erlaubt! (Divisor: %)', b;
   END IF;
   RETURN a::DECIMAL / b;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Testen: Erfolg
SELECT divide(10, 2) as result
```

#	result
1	5.0000000000000000

1 rows

```
SELECT divide(100, 4) as result
```

#	result
1	25.0000000000000000

1 rows

```
-- Testen: Fehler
SELECT divide(10, 0) as result
```

Division durch Null ist nicht erlaubt! (Divisor: 0)

Probieren Sie die letzte Zeile aus – Sie sehen eine klare Fehlermeldung! Das ist besser als ein kryptischer Datenbankfehler.

Praxisbeispiel: Preisberechnung

Demo 6: Preisberechnung mit MwSt.

Kombinieren wir alles Gelernte in einem realistischen Beispiel: Gesamtpreis mit Steuer berechnen.

```
1  CREATE FUNCTION calculate_total(price DECIMAL, tax_rate DECIMAL)
2  RETURNS DECIMAL AS $$
3  BEGIN
4      IF price < 0 THEN
5          RAISE EXCEPTION 'Preis kann nicht negativ sein: %', price;
6      END IF;
```

```
7
8       IF tax_rate < 0 OR tax_rate > 1 THEN
9           RAISE EXCEPTION 'Steuersatz muss zwischen 0 und 1 liegen: %',
              tax_rate;
10      END IF;
11
12      RETURN price * (1 + tax_rate);
13  END;
14  $$ LANGUAGE plpgsql;
15
16  -- Testen mit verschiedenen Szenarien
17  SELECT calculate_total(100, 0.19) as brutto;    -- Deutschland: 19% M
18  SELECT calculate_total(50, 0.07) as brutto;     -- Ermäßigt: 7%
19  SELECT calculate_total(200, 0) as brutto;       -- Steuerfrei
20
21  -- Fehler provozieren:
22  -- SELECT calculate_total(-10, 0.19) as brutto;  -- ❌ Negativer Prei
23  -- SELECT calculate_total(100, 1.5) as brutto;   -- ❌ Ungültiger Ste
```

```
CREATE FUNCTION calculate_total(price DECIMAL, tax_rate DECIMAL)
RETURNS DECIMAL AS $$
BEGIN
    IF price < 0 THEN
        RAISE EXCEPTION 'Preis kann nicht negativ sein: %', price;
    END IF;

    IF tax_rate < 0 OR tax_rate > 1 THEN
        RAISE EXCEPTION 'Steuersatz muss zwischen 0 und 1 liegen: %', tax_rate;
    END IF;

    RETURN price * (1 + tax_rate);
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Testen mit verschiedenen Szenarien
SELECT calculate_total(100, 0.19) as brutto
```

#	brutto
1	119.00

1 rows

```
-- Deutschland: 19% MwSt
SELECT calculate_total(50, 0.07) as brutto
```

#	brutto
1	53.50

1 rows

```
-- Ermäßigt: 7%
SELECT calculate_total(200, 0) as brutto
```

#	brutto
1	200

1 rows

```
-- Steuerfrei

-- Fehler provozieren:
-- SELECT calculate_total(-10, 0.19) as brutto;  -- ❌ Negativer Preis
-- SELECT calculate_total(100, 1.5) as brutto;   -- ❌ Ungültiger Steuersatz
```

ok

Perfekt! Jetzt können Sie solide Functions schreiben. Aber was, wenn Sie wollen, dass Code automatisch ausgeführt wird – ohne dass jemand die Funktion aufruft? Genau dafür gibt es Trigger!

Teil 2: Trigger

Trigger sind das Automatisierungs-Werkzeug der Datenbank. Sie „triggern“ – werden ausgelöst – bei bestimmten Events: INSERT, UPDATE oder DELETE. Denken Sie an Event-Listener in JavaScript, aber auf Datenbankebene.

Was sind Trigger?

Definition:

Ein Trigger ist eine Funktion, die automatisch ausgeführt wird, wenn ein bestimmtes Event auf einer Tabelle passiert.

Komponenten:

1. **Trigger-Function:** Eine spezielle Function mit `RETURNS TRIGGER`
2. **Trigger:** Verbindet die Function mit einer Tabelle und einem Event

Syntax:

```
-- 1. Function erstellen
CREATE FUNCTION trigger_function()
RETURNS TRIGGER AS $$
BEGIN
    -- Code hier
    RETURN NEW;  -- oder OLD oder NULL
END;
$$ LANGUAGE plpgsql;

-- 2. Trigger erstellen
CREATE TRIGGER trigger_name
BEFORE UPDATE ON table_name
FOR EACH ROW
EXECUTE FUNCTION trigger_function();
```

Besonderheiten von Trigger-Functions

Trigger-Functions sind anders als normale Functions:

Spezielle Variablen:

Variable	Typ	Beschreibung	Verfügbar bei
`NEW`	RECORD	Die neue Zeile	INSERT, UPDATE
`OLD`	RECORD	Die alte Zeile	UPDATE, DELETE

Beispiel:

```
CREATE FUNCTION my_trigger()
RETURNS TRIGGER AS $$
BEGIN
    -- Bei INSERT: nur NEW verfügbar
    -- Bei UPDATE: OLD und NEW verfügbar
    -- Bei DELETE: nur OLD verfügbar

    RAISE NOTICE 'Alte Zeile: %, Neue Zeile: %', OLD, NEW;

    RETURN NEW;  -- Gibt die (ggf. modifizierte) Zeile zurück
END;
$$ LANGUAGE plpgsql;
```

RETURN-Werte bei BEFORE-Triggern

Bei BEFORE-Triggern ist der RETURN-Wert wichtig:

RETURN	Bedeutung
`RETURN NEW;`	Änderungen übernehmen (bei INSERT/UPDATE)
`RETURN OLD;`	Ursprüngliche Werte behalten (bei UPDATE)
`RETURN NULL;`	Operation abbrechen! (bei DELETE: Zeile wird NICHT gelöscht)

Bei AFTER-Triggern: RETURN-Wert wird ignoriert, `RETURN NULL;` ist üblich.

CREATE TRIGGER Syntax

So erstellen Sie einen Trigger:

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE [ OR ... ] }
ON table_name
FOR EACH ROW
EXECUTE FUNCTION function_name();
```

Optionen:

- `BEFORE` – Trigger läuft VOR der Operation (kann Daten ändern oder Operation abbrechen)
- `AFTER` – Trigger läuft NACH der Operation (kann nicht mehr eingreifen)
- `FOR EACH ROW` – Trigger wird für jede betroffene Zeile ausgeführt
- Mehrere Events: `BEFORE INSERT OR UPDATE OR DELETE`

Genug Theorie – schauen wir uns vier praktische Beispiele an, die Sie sofort nutzen können!

Demo 7: Automatische Timestamps

Das häufigste Use Case: Timestamp-Felder automatisch aktualisieren.

Schritt 1: Tabelle vorbereiten

Zuerst erstellen wir eine Produkte-Tabelle mit Timestamp-Feldern.

```sql
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Testdaten einfügen
INSERT INTO products (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99);

-- Ausgangszustand
SELECT id, name, price, created_at, updated_at FROM products;
```

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
)
```

ok

```
-- Testdaten einfügen
INSERT INTO products (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99)
```

ok

```
-- Ausgangszustand
SELECT id, name, price, created_at, updated_at FROM products
```

#	id	name	price	created_at	updated_at
1	1	Laptop	999.99	2026-02-13T10:42:43.529Z	2026-02-13T10:42:43.529Z
2	2	Maus	29.99	2026-02-13T10:42:43.529Z	2026-02-13T10:42:43.529Z

2 rows

Schritt 2: Trigger-Function & Trigger erstellen

Jetzt die Magie: Eine Function, die updated_at automatisch setzt.

```sql
-- Function: Setzt updated_at auf NOW()
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger: Wird bei jedem UPDATE ausgeführt
CREATE TRIGGER set_updated_at
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();

-- Bestätigung
SELECT 'Trigger erfolgreich erstellt!' as status;
```

```
-- Function: Setzt updated_at auf NOW()
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
   NEW.updated_at = NOW();
   RETURN NEW;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Trigger: Wird bei jedem UPDATE ausgeführt
CREATE TRIGGER set_updated_at
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_timestamp()
```

ok

```
-- Bestätigung
SELECT 'Trigger erfolgreich erstellt!' as status
```

#	status
1	Trigger erfolgreich erstellt!

1 rows

Schritt 3: Testen

Jetzt ändern wir Daten und schauen, ob updated_at automatisch aktualisiert wird.

```sql
-- Kurze Pause simulieren (damit Zeitunterschied sichtbar ist)
SELECT pg_sleep(1);

-- Preis ändern
UPDATE products
SET price = 899.99
WHERE name = 'Laptop';

-- Ergebnis prüfen
SELECT
    name,
    price,
    created_at,
    updated_at,
    (updated_at > created_at) as timestamp_updated
FROM products;
```

```
-- Kurze Pause simulieren (damit Zeitunterschied sichtbar ist)
SELECT pg_sleep(1)
```

#	pg_sleep
1	

1 rows

```
-- Preis ändern
UPDATE products
SET price = 899.99
WHERE name = 'Laptop'
```

ok

```
-- Ergebnis prüfen
SELECT
   name,
   price,
   created_at,
   updated_at,
   (updated_at > created_at) as timestamp_updated
FROM products
```

#	name	price	created_at	updated_at	timestamp_updated
1	Maus	29.99	2026-02-13T10:42:43.529Z	2026-02-13T10:42:43.529Z	**false**
2	Laptop	899.99	2026-02-13T10:42:43.529Z	2026-02-13T10:42:45.833Z	**true**

2 rows

Perfekt! Das updated_at-Feld wurde automatisch aktualisiert – ohne dass wir es in der UPDATE-Query angeben mussten. Das funktioniert jetzt für jedes Update, egal aus welcher Anwendung!

Demo 8: Audit-Logging

Zweitens: Änderungen protokollieren für Compliance und Nachvollziehbarkeit.

Schritt 1: Tabellen vorbereiten

Wir brauchen eine Audit-Tabelle, um Änderungen zu protokollieren.

```
1  -- Produkte-Tabelle
2  CREATE TABLE products_audit_demo (
3      id SERIAL PRIMARY KEY,
4      name TEXT
```

```
    name TEXT,
    price DECIMAL(10, 2)
);

-- Audit-Tabelle
CREATE TABLE products_audit_log (
    audit_id SERIAL PRIMARY KEY,
    product_id INT,
    old_price DECIMAL(10, 2),
    new_price DECIMAL(10, 2),
    changed_at TIMESTAMP DEFAULT NOW()
);

-- Testdaten
INSERT INTO products_audit_demo (name, price) VALUES ('Laptop', 999.9

-- Ausgangszustand
SELECT * FROM products_audit_demo;
SELECT * FROM products_audit_log;  -- Noch leer
```

```
-- Produkte-Tabelle
CREATE TABLE products_audit_demo (
    id SERIAL PRIMARY KEY,
    name TEXT,
    price DECIMAL(10, 2)
)
```

ok

```
-- Audit-Tabelle
CREATE TABLE products_audit_log (
    audit_id SERIAL PRIMARY KEY,
    product_id INT,
    old_price DECIMAL(10, 2),
    new_price DECIMAL(10, 2),
    changed_at TIMESTAMP DEFAULT NOW()
)
```

ok

```
-- Testdaten
INSERT INTO products_audit_demo (name, price) VALUES ('Laptop', 999.99)
```

ok

```
-- Ausgangszustand
SELECT * FROM products_audit_demo
```

#	id	name	price
1	1	Laptop	999.99

1 rows

```
SELECT * FROM products_audit_log
```

#	audit_id	product_id	old_price	new_price	changed_at

0 rows

```
-- Noch leer
```

ok

Schritt 2: Audit-Trigger erstellen

Function, die Preisänderungen protokolliert:

```sql
-- Function: Protokolliert Preisänderungen
CREATE OR REPLACE FUNCTION log_price_change()
RETURNS TRIGGER AS $$
BEGIN
    -- Nur protokollieren, wenn sich der Preis tatsächlich geändert h
    IF OLD.price IS DISTINCT FROM NEW.price THEN
        INSERT INTO products_audit_log (product_id, old_price, new_pr
        VALUES (NEW.id, OLD.price, NEW.price);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger: Wird NACH jedem UPDATE ausgeführt
CREATE TRIGGER audit_price_changes
AFTER UPDATE ON products_audit_demo
FOR EACH ROW
EXECUTE FUNCTION log_price_change();

SELECT 'Audit-Trigger erstellt!' as status;
```

```
-- Function: Protokolliert Preisänderungen
CREATE OR REPLACE FUNCTION log_price_change()
RETURNS TRIGGER AS $$
BEGIN
    -- Nur protokollieren, wenn sich der Preis tatsächlich geändert hat
    IF OLD.price IS DISTINCT FROM NEW.price THEN
        INSERT INTO products_audit_log (product_id, old_price, new_price)
        VALUES (NEW.id, OLD.price, NEW.price);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Trigger: Wird NACH jedem UPDATE ausgeführt
CREATE TRIGGER audit_price_changes
AFTER UPDATE ON products_audit_demo
FOR EACH ROW
EXECUTE FUNCTION log_price_change()
```

ok

```
SELECT 'Audit-Trigger erstellt!' as status
```

#	status
1	Audit-Trigger erstellt!

1 rows

Schritt 3: Testen

Jetzt ändern wir den Preis mehrmals und schauen ins Audit-Log.

```
1  -- Mehrere Preisänderungen
2  UPDATE products_audit_demo SET price = 899.99 WHERE name = 'Laptop';
3  UPDATE products_audit_demo SET price = 799.99 WHERE name = 'Laptop';
4  UPDATE products_audit_demo SET price = 849.99 WHERE name = 'Laptop';
5
6  -- Aktueller Zustand
7  SELECT * FROM products_audit_demo;
8
9  -- Audit-Log: Alle Änderungen protokolliert!
10 SELECT
11     audit_id,
12     product_id,
13     old_price,
```

```
    new_price,
    old_price - new_price as price_change,
    changed_at
FROM products_audit_log
ORDER BY changed_at;
```

```
-- Mehrere Preisänderungen
UPDATE products_audit_demo SET price = 899.99 WHERE name = 'Laptop'
```

ok

```
UPDATE products_audit_demo SET price = 799.99 WHERE name = 'Laptop'
```

ok

```
UPDATE products_audit_demo SET price = 849.99 WHERE name = 'Laptop'
```

ok

```
-- Aktueller Zustand
SELECT * FROM products_audit_demo
```

#	id	name	price
1	1	Laptop	849.99

1 rows

```
-- Audit-Log: Alle Änderungen protokolliert!
SELECT
    audit_id,
    product_id,
    old_price,
    new_price,
    old_price - new_price as price_change,
    changed_at
FROM products_audit_log
ORDER BY changed_at
```

#	audit_id	product_id	old_price	new_price	price_change	changed_at
1	1	1	999.99	899.99	100.00	2026-02-13T10:42:48.977Z
2	2	1	899.99	799.99	100.00	2026-02-13T10:42:48.980Z
3	3	1	799.99	849.99	-50.00	2026-02-13T10:42:48.980Z

3 rows

Exzellent! Jede Preisänderung wurde automatisch protokolliert. Das ist perfekt für Compliance-Anforderungen – die Anwendung kann das Logging nicht „vergessen“.

Demo 9: Validierung

Drittens: Datenintegrität mit Triggern erzwingen – z.B. negative Preise verhindern.

Schritt 1: Tabelle & Trigger erstellen

Wir erstellen eine Tabelle und einen Trigger, der negative Preise verhindert.

```sql
-- Tabelle
CREATE TABLE products_validation (
    id SERIAL PRIMARY KEY,
    name TEXT,
    price DECIMAL(10, 2)
);

-- Function: Prüft, ob Preis gültig ist
CREATE OR REPLACE FUNCTION prevent_negative_price()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.price < 0 THEN
        RAISE EXCEPTION 'Preis % ist ungültig (negativ)!', NEW.price;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger: Läuft bei INSERT und UPDATE
CREATE TRIGGER check_price
BEFORE INSERT OR UPDATE ON products_validation
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_price();

SELECT 'Validierungs-Trigger erstellt!' as status;
```

```
-- Tabelle
CREATE TABLE products_validation (
    id SERIAL PRIMARY KEY,
    name TEXT,
    price DECIMAL(10, 2)
)
```

ok

```
-- Function: Prüft, ob Preis gültig ist
CREATE OR REPLACE FUNCTION prevent_negative_price()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.price < 0 THEN
        RAISE EXCEPTION 'Preis % ist ungültig (negativ)!', NEW.price;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Trigger: Läuft bei INSERT und UPDATE
CREATE TRIGGER check_price
BEFORE INSERT OR UPDATE ON products_validation
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_price()
```

ok

```
SELECT 'Validierungs-Trigger erstellt!' as status
```

#	status
1	Validierungs-Trigger erstellt!

1 rows

Schritt 2: Erfolgreiche Einfügungen

Zuerst testen wir mit gültigen Daten:

```
1  -- Gültige Inserts
2  INSERT INTO products_validation (name, price) VALUES ('Laptop', 999.99
3  INSERT INTO products_validation (name, price) VALUES ('Maus', 29.99);
4  INSERT INTO products_validation (name, price) VALUES ('Gratis-Ebook',
5
6  -- Alles funktioniert
```

```
7  SELECT * FROM products_validation;
```

-- Gültige Inserts
INSERT INTO products_validation (name, price) VALUES ('Laptop', 999.99)

ok

INSERT INTO products_validation (name, price) VALUES ('Maus', 29.99)

ok

INSERT INTO products_validation (name, price) VALUES ('Gratis-Ebook', 0.00)

ok

-- Alles funktioniert
SELECT * FROM products_validation

#	id	name	price
1	1	Laptop	999.99
2	2	Maus	29.99
3	3	Gratis-Ebook	0.00

3 rows

Schritt 3: Ungültige Daten provozieren

Jetzt versuchen wir, einen negativen Preis einzufügen:

```
1  -- Dieser Versuch schlägt fehl!
2  INSERT INTO products_validation (name, price) VALUES ('Fehlerhaft', -1
3
4  -- ❌ ERROR: Preis -10.00 ist ungültig (negativ)!
```

-- Dieser Versuch schlägt fehl!
INSERT INTO products_validation (name, price) VALUES ('Fehlerhaft', -10.00)

Preis -10.00 ist ungültig (negativ)!

Perfekt! Der Trigger hat die ungültige Operation verhindert. Die Anwendung kann diese Regel nicht umgehen – sie ist in der Datenbank verankert.

Demo 10: Soft Delete mit Views & INSTEAD OF Trigger

Viertens: Löschen, ohne wirklich zu löschen – für Wiederherstellung und Audit-Zwecke. Diesmal mit einem eleganten Twist: Die Anwendung arbeitet nur mit einer View und weiß gar nicht, dass Soft Delete passiert!

Schritt 1: Basis-Tabelle mit Soft-Delete-Flag

Wir erstellen die eigentliche Produkte-Tabelle mit einem deleted_at Feld:

```sql
-- Basis-Tabelle (kennt die Anwendung nicht!)
CREATE TABLE products_base (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    deleted_at TIMESTAMP  -- NULL = aktiv, Timestamp = gelöscht
);

-- Testdaten
INSERT INTO products_base (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99),
    ('Tastatur', 79.99);

-- Alle Daten (inkl. deleted_at)
SELECT * FROM products_base;
```

```
-- Basis-Tabelle (kennt die Anwendung nicht!)
CREATE TABLE products_base (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    deleted_at TIMESTAMP  -- NULL = aktiv, Timestamp = gelöscht
)
```

ok

```
-- Testdaten
INSERT INTO products_base (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99),
    ('Tastatur', 79.99)
```

ok

```
-- Alle Daten (inkl. deleted_at)
SELECT * FROM products_base
```

#	id	name	price	created_at	deleted_at
1	1	Laptop	999.99	2026-02-13T10:42:55.694Z	*null*
2	2	Maus	29.99	2026-02-13T10:42:55.694Z	*null*
3	3	Tastatur	79.99	2026-02-13T10:42:55.694Z	*null*

3 rows

Schritt 2: View für aktive Produkte

Die Anwendung arbeitet nur mit dieser View – sie zeigt nur aktive Produkte:

```sql
-- View: Die "öffentliche" Schnittstelle zur Datenbank
CREATE VIEW products AS
SELECT id, name, price, created_at
FROM products_base
WHERE deleted_at IS NULL;  -- Filter: nur aktive Produkte

-- Anwendung sieht nur diese View
SELECT * FROM products;
```

```
-- View: Die "öffentliche" Schnittstelle zur Datenbank
CREATE VIEW products AS
SELECT id, name, price, created_at
FROM products_base
WHERE deleted_at IS NULL
```

ok

```
-- Filter: nur aktive Produkte

-- Anwendung sieht nur diese View
SELECT * FROM products
```

#	id	name	price	created_at
1	1	Laptop	999.99	2026-02-13T10:42:55.694Z
2	2	Maus	29.99	2026-02-13T10:42:55.694Z
3	3	Tastatur	79.99	2026-02-13T10:42:55.694Z

3 rows

—{12}– Beachten Sie: Die View zeigt das deleted_at Feld gar nicht – die Anwendung weiß nichts von Soft Delete!

Schritt 3: INSTEAD OF Trigger auf der View

Jetzt kommt die Magie: Ein Trigger auf der View, der DELETE-Operationen abfängt:

```
-- Function: Führt Soft Delete auf der Basis-Tabelle aus
CREATE OR REPLACE FUNCTION soft_delete_via_view()
RETURNS TRIGGER AS $$
BEGIN
    -- Setzt deleted_at auf der echten Tabelle
    UPDATE products_base
    SET deleted_at = NOW()
    WHERE id = OLD.id;

    -- RETURN OLD bei INSTEAD OF Triggern
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

-- INSTEAD OF Trigger: Ersetzt DELETE auf der View
CREATE TRIGGER soft_delete_products
INSTEAD OF DELETE ON products
FOR EACH ROW
EXECUTE FUNCTION soft_delete_via_view();

SELECT 'Soft-Delete-Trigger auf View erstellt!' as status;
```

```
-- Function: Führt Soft Delete auf der Basis-Tabelle aus
CREATE OR REPLACE FUNCTION soft_delete_via_view()
RETURNS TRIGGER AS $$
BEGIN
    -- Setzt deleted_at auf der echten Tabelle
    UPDATE products_base
    SET deleted_at = NOW()
    WHERE id = OLD.id;

    -- RETURN OLD bei INSTEAD OF Triggern
    RETURN OLD;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- INSTEAD OF Trigger: Ersetzt DELETE auf der View
CREATE TRIGGER soft_delete_products
INSTEAD OF DELETE ON products
FOR EACH ROW
EXECUTE FUNCTION soft_delete_via_view()
```

ok

```
SELECT 'Soft-Delete-Trigger auf View erstellt!' as status
```

#	status
1	Soft-Delete-Trigger auf View erstellt!

1 rows

INSTEAD OF Trigger funktionieren nur auf Views und ersetzen die Operation komplett. Perfekt für unseren Use Case!

Schritt 4: „Löschen“ über die View

Die Anwendung „löscht“ ein Produkt – aber es wird nur markiert:

```
-- Anwendung löscht über die View (weiß nichts von Soft Delete!)
DELETE FROM products WHERE name = 'Maus';

-- View zeigt nur noch aktive Produkte
SELECT 'Aktive Produkte (View):' as info;
SELECT * FROM products;

-- Basis-Tabelle zeigt ALLE Produkte (inkl. deleted_at)
SELECT 'Alle Produkte (Basis-Tabelle):' as info;
```

```sql
SELECT
    id,
    name,
    price,
    deleted_at,
    CASE
        WHEN deleted_at IS NULL THEN '✅ Aktiv'
        ELSE '❌ Gelöscht'
    END as status
FROM products_base
ORDER BY id;
```

```
-- Anwendung löscht über die View (weiß nichts von Soft Delete!)
DELETE FROM products WHERE name = 'Maus'
```

ok

```
-- View zeigt nur noch aktive Produkte
SELECT 'Aktive Produkte (View):' as info
```

#	info
1	Aktive Produkte (View):

1 rows

```
SELECT * FROM products
```

#	id	name	price	created_at
1	1	Laptop	999.99	2026-02-13T10:42:55.694Z
2	3	Tastatur	79.99	2026-02-13T10:42:55.694Z

2 rows

```
-- Basis-Tabelle zeigt ALLE Produkte (inkl. deleted_at)
SELECT 'Alle Produkte (Basis-Tabelle):' as info
```

#	info
1	Alle Produkte (Basis-Tabelle):

1 rows

```
SELECT
    id,
    name,
    price,
    deleted_at,
    CASE
        WHEN deleted_at IS NULL THEN '✅ Aktiv'
        ELSE '❌ Gelöscht'
    END as status
FROM products_base
ORDER BY id
```

#	id	name	price	deleted_at	status
1	1	Laptop	999.99	*null*	✅ Aktiv
2	2	Maus	29.99	2026-02-13T10:42:58.015Z	❌ Gelöscht
3	3	Tastatur	79.99	*null*	✅ Aktiv

3 rows

Brilliant! Die Maus ist aus der View verschwunden – aber in der Basis-Tabelle noch vorhanden mit gesetztem deleted_at Timestamp. Die Anwendung merkt nichts von der Implementierung!

Schritt 5: Wiederherstellung

—{{ß}}– Gelöschte Produkte können einfach wiederhergestellt werden:

```
1  -- Admin-Funktion: Produkt wiederherstellen
2  UPDATE products_base
3  SET deleted_at = NULL
4  WHERE name = 'Maus';
5
6  -- View zeigt das Produkt wieder!
7  SELECT * FROM products;
```

-- Admin-Funktion: Produkt wiederherstellen
UPDATE products_base
SET deleted_at = NULL
WHERE name = 'Maus'

ok

-- View zeigt das Produkt wieder!
SELECT * FROM products

#	id	name	price	created_at
1	1	Laptop	999.99	2026-02-13T10:42:55.694Z
2	3	Tastatur	79.99	2026-02-13T10:42:55.694Z
3	2	Maus	29.99	2026-02-13T10:42:55.694Z

3 rows

Perfekt! Durch die View-Abstraktion haben Sie eine saubere Trennung: Die Anwendung arbeitet mit der View, Admins können auf die Basis-Tabelle zugreifen.

Warum ist das elegant?

Schauen wir uns die Vorteile an:

Vorteile dieser Architektur:

Aspekt	Ohne View	Mit View + INSTEAD OF Trigger
Anwendungscode	Muss Soft Delete implementieren	Arbeitet normal mit DELETE
Komplexität	Verteilt über viele Stellen	Zentralisiert in der DB
Konsistenz	Entwickler können es vergessen	Automatisch garantiert
Wiederherstellung	Muss explizit implementiert werden	Einfaches UPDATE auf Basis-Tabelle
Migration	Anwendung muss angepasst werden	Transparent – keine Code-Änderung
Testen	Schwierig (überall prüfen)	Einfach (nur View testen)

Anwendungscode-Vergleich:

```
// Ohne View: Anwendung muss Soft Delete kennen
await db.query(
  'UPDATE products SET deleted_at = NOW() WHERE id = $1',
  [productId]
);

// Mit View: Anwendung nutzt normales DELETE
await db.query(
  'DELETE FROM products WHERE id = $1',
  [productId]
);
// ✅ Trigger macht den Rest - transparent!
```

Best Practice: Diese Architektur nennt sich **Database Abstraction Layer**. Die View ist die öffentliche API, die Implementierung dahinter kann sich ändern, ohne die Anwendung anzufassen.

Gefahren & Best Practices

Trigger sind mächtig – aber mit großer Macht kommt große Verantwortung! Schauen wir uns potenzielle Probleme an.

Gefahr 1: Trigger-Kaskaden

Das größte Problem: Trigger, die andere Trigger auslösen – eine Kettenreaktion!

Szenario:

```
Trigger A (on products)
  → UPDATE inventory
    → Trigger B (on inventory)
      → INSERT audit_log
        → Trigger C (on audit_log)
          → UPDATE statistics
            → Trigger D (on statistics)
              → ... 💥
```

Problem:

- ❌ Schwer zu debuggen
- ❌ Performance-Einbruch
- ❌ Risiko von Endlosschleifen
- ❌ Unvorhersehbares Verhalten

Lösung:

```
-- NIEMALS in einem Trigger weitere Trigger auslösen!
-- Stattdessen: Komplexe Logik in eine Funktion auslagern
CREATE FUNCTION process_order()
RETURNS VOID AS $$
BEGIN
    -- Alle Operationen explizit hier
    UPDATE inventory ...;
    INSERT INTO audit_log ...;
    UPDATE statistics ...;
END;
$$ LANGUAGE plpgsql;
```

Gefahr 2: Performance-Impact

Trigger laufen bei JEDER Operation – auch bei BULK Inserts!

Problem:

```
-- BULK INSERT von 100.000 Zeilen
INSERT INTO products SELECT * FROM imported_data;

-- Wenn ein Trigger existiert:
-- → 100.000× Trigger-Ausführung!
-- → Kann Minuten statt Sekunden dauern
```

Lösung:

```sql
-- Trigger temporär deaktivieren (PostgreSQL)
ALTER TABLE products DISABLE TRIGGER set_updated_at;

-- BULK Operation
INSERT INTO products SELECT * FROM imported_data;

-- Trigger wieder aktivieren
ALTER TABLE products ENABLE TRIGGER set_updated_at;
```

Best Practice: Überlegen Sie, ob ein Batch-Job statt Trigger sinnvoller ist!

Gefahr 3: Debugging-Schwierigkeiten

Trigger sind unsichtbar für die Anwendung – Fehler sind schwer zu finden.

Problem:

```
// Anwendungscode
await db.query('UPDATE products SET price = 99.99 WHERE id = 1');

// ❓ Plötzlich ist die Performance schlecht
// ❓ Plötzlich gibt es unerwartete Änderungen in anderen Tabellen
// ❓ Die Anwendung weiß nicht, dass Trigger existieren!
```

Lösung:

1. **Dokumentation:** Kommentiere alle Trigger im Schema-Script
2. **Naming Convention:** `trigger_<table>_<event>_<action>`
3. **Logging:** RAISE NOTICE in Triggern für Debugging
4. **Monitoring:** Query-Performance überwachen

```sql
CREATE TRIGGER trigger_products_after_update_audit
AFTER UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION log_price_change();

-- Name verrät: Tabelle = products, Event = update, Aktion = audit
```

Best Practice 1: Trigger nur wenn nötig

Viele Anforderungen können einfacher gelöst werden!

Anforderung	❌ Trigger	✅ Bessere Lösung
Validierung	`CREATE TRIGGER check_price...`	`CHECK (price >= 0)`
Default-Werte	`CREATE TRIGGER set_default...`	`DEFAULT NOW()`
Ref. Integrität	`CREATE TRIGGER check_fk...`	`FOREIGN KEY`
Timestamps	✅ Trigger ist OK	Oder: `DEFAULT NOW()` + Trigger für UPDATE
Audit-Logging	✅ Trigger ist ideal	Keine Alternative
Soft Delete	✅ Trigger ist gut	Oder: App-seitig

Faustregel: Nutze deklarative Constraints wo möglich, Trigger nur wenn nötig!

Best Practice 2: BEFORE vs. AFTER

Wann welchen Trigger-Typ nutzen?

Use Case	BEFORE	AFTER
Daten ändern (z.B. Timestamps)	✅ Ja	❌ Zu spät
Validierung (z.B. negative Preise)	✅ Ja	❌ Zu spät
Operation abbrechen	✅ RETURN NULL	❌ Nicht möglich
Audit-Logging	⚠️ Möglich	✅ Besser (Änderung ist garantiert committed)
Andere Tabellen ändern	⚠️ Möglich	✅ Besser (Hauptoperation ist fertig)

Faustregel:

- **BEFORE** für Änderungen an der aktuellen Zeile
- **AFTER** für Änderungen an anderen Tabellen oder Logging

Best Practice 3: Testen, testen, testen!

Trigger sind Code – und Code muss getestet werden!

Setup: Testumgebung vorbereiten

Zuerst erstellen wir eine Testumgebung mit Produkten-Tabelle und allen Triggern:

```sql
-- Tabelle mit Timestamp-Feldern und Validierung
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Function 1: Timestamp automatisch aktualisieren
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger 1: Setzt updated_at bei jedem UPDATE
CREATE TRIGGER set_updated_at
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();

-- Function 2: Negative Preise verhindern
CREATE OR REPLACE FUNCTION prevent_negative_price()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.price < 0 THEN
        RAISE EXCEPTION 'Preis % ist ungültig (negativ)!', NEW.price;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger 2: Validierung bei INSERT und UPDATE
CREATE TRIGGER check_price
BEFORE INSERT OR UPDATE ON products
FOR EACH ROW
```

```
EXECUTE FUNCTION prevent_negative_price();

-- Testdaten
INSERT INTO products (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99);

-- Status
SELECT 'Test-Umgebung erfolgreich erstellt!' as status;
SELECT * FROM products;
```

```
-- Tabelle mit Timestamp-Feldern und Validierung
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
)
```

ok

```
-- Function 1: Timestamp automatisch aktualisieren
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Trigger 1: Setzt updated_at bei jedem UPDATE
CREATE TRIGGER set_updated_at
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_timestamp()
```

ok

```
-- Function 2: Negative Preise verhindern
CREATE OR REPLACE FUNCTION prevent_negative_price()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.price < 0 THEN
        RAISE EXCEPTION 'Preis % ist ungültig (negativ)!', NEW.price;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql
```

ok

```
-- Trigger 2: Validierung bei INSERT und UPDATE
CREATE TRIGGER check_price
BEFORE INSERT OR UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_price()
```

ok

```
-- Testdaten
INSERT INTO products (name, price) VALUES
    ('Laptop', 999.99),
    ('Maus', 29.99)
```

ok

```
-- Status
SELECT 'Test-Umgebung erfolgreich erstellt!' as status
```

#	status
1	Test-Umgebung erfolgreich erstellt!

1 rows

```
SELECT * FROM products
```

#	id	name	price	created_at	updated_at
1	1	Laptop	999.99	2026-02-13T10:43:08.781Z	2026-02-13T10:43:08.781Z
2	2	Maus	29.99	2026-02-13T10:43:08.781Z	2026-02-13T10:43:08.781Z

2 rows

Test 1: Erfolgreicher Fall (Timestamp-Update)

Testen wir, ob der Timestamp-Trigger korrekt funktioniert. Wir nutzen eine Transaktion mit ROLLBACK, um die Testdaten nicht dauerhaft zu ändern:

```
-- Test 1: Erfolgreicher Fall
-- Erwartung: updated_at wird automatisch aktualisiert

BEGIN;  -- Transaktion starten

-- Vor dem Update
SELECT
    name,
    price,
    created_at,
    updated_at,
    'BEFORE UPDATE' as moment
FROM products WHERE name = 'Laptop';

-- Kurze Pause für sichtbaren Zeitunterschied
SELECT pg_sleep(0.5);

```

```sql
-- Update durchführen
UPDATE products SET price = 899.99 WHERE name = 'Laptop';

-- Nach dem Update: updated_at sollte NACH created_at liegen
SELECT
    name,
    price,
    created_at,
    updated_at,
    'AFTER UPDATE' as moment,
    (updated_at > created_at) as timestamp_wurde_aktualisiert
FROM products WHERE name = 'Laptop';

-- ✅ Test erfolgreich wenn: timestamp_wurde_aktualisiert = true

ROLLBACK;  -- Änderungen verwerfen, Daten bleiben unverändert

-- Prüfen: Daten sind wieder im Originalzustand
SELECT 'Nach ROLLBACK:' as info, * FROM products WHERE name = 'Laptop
```

```
-- Test 1: Erfolgreicher Fall
-- Erwartung: updated_at wird automatisch aktualisiert

BEGIN
```

ok

```
-- Transaktion starten

-- Vor dem Update
SELECT
    name,
    price,
    created_at,
    updated_at,
    'BEFORE UPDATE' as moment
FROM products WHERE name = 'Laptop'
```

#	name	price	created_at	updated_at	moment
1	Laptop	999.99	2026-02-13T10:43:08.781Z	2026-02-13T10:43:08.781Z	BEFORE UPDATE

1 rows

```
-- Kurze Pause für sichtbaren Zeitunterschied
SELECT pg_sleep(0.5)
```

#	pg_sleep
1	

1 rows

```
-- Update durchführen
UPDATE products SET price = 899.99 WHERE name = 'Laptop'
```

ok

```
-- Nach dem Update: updated_at sollte NACH created_at liegen
SELECT
    name,
    price,
    created_at,
    updated_at,
    'AFTER UPDATE' as moment,
    (updated_at > created_at) as timestamp_wurde_aktualisiert
FROM products WHERE name = 'Laptop'
```

#	name	price	created_at	updated_at	moment	timestamp_wurde_ak
1	Laptop	899.99	2026-02-13T10:43:08.781Z	2026-02-13T10:43:09.153Z	AFTER UPDATE	true

1 rows

```
-- ✅ Test erfolgreich wenn: timestamp_wurde_aktualisiert = true

ROLLBACK
```

ok

```
-- Änderungen verwerfen, Daten bleiben unverändert

-- Prüfen: Daten sind wieder im Originalzustand
SELECT 'Nach ROLLBACK:' as info, * FROM products WHERE name = 'Laptop'
```

#	info	id	name	price	created_at	updated_at
1	Nach ROLLBACK:	1	Laptop	999.99	2026-02-13T10:43:08.781Z	2026-02-13T10:43:08.781Z

1 rows

Test 2: Fehlerfall (Negative Preise)

Jetzt testen wir die Validierung – negative Preise müssen verhindert werden. Die Transaktion wird automatisch zurückgerollt, wenn ein Fehler auftritt:

```
1  -- Test 2: Fehlerfall
2  -- Erwartung: INSERT mit negativem Preis wird abgelehnt
3
4  BEGIN;  -- Transaktion starten
5
6  -- Anzahl Produkte vor dem Test
7  SELECT 'Vor dem Test:' as info, COUNT(*) as anzahl_produkte FROM prod
8
9  -- Dieser Versuch MUSS fehlschlagen:
10 INSERT INTO products (name, price) VALUES ('Fehlerprodukt', -10.00);
11
12 -- ❌ Erwartete Fehlermeldung: "Preis -10.00 ist ungültig (negativ)!"
13 -- ❌ Diese Zeile wird NICHT erreicht, da vorher eine Exception gewor
     wird
14
15 ROLLBACK;  -- Wird nur bei manuellem Aufruf erreicht
16
17 -- Nach dem Fehler: Prüfen, dass keine Daten eingefügt wurden
18 SELECT 'Nach dem fehlgeschlagenen INSERT:' as info, COUNT(*) as
     anzahl_produkte FROM products;
```

```
19
20 -- ✅ Test erfolgreich wenn: Exception wird geworfen UND anzahl_produ
     bleibt gleich
```

-- Test 2: Fehlerfall
-- Erwartung: INSERT mit negativem Preis wird abgelehnt

BEGIN

ok

-- Transaktion starten

-- Anzahl Produkte vor dem Test
SELECT 'Vor dem Test:' as info, COUNT(*) as anzahl_produkte FROM products

#	info	anzahl_produkte
1	Vor dem Test:	2

1 rows

-- Dieser Versuch MUSS fehlschlagen:
INSERT INTO products (name, price) VALUES ('Fehlerprodukt', -10.00)

Preis -10.00 ist ungültig (negativ)!

Test 3: Edge Cases (NULL-Werte)

Edge Cases sind wichtig – was passiert mit NULL? Auch hier nutzen wir eine Transaktion:

```
1  -- Test 3: Edge Cases
2  -- Erwartung: NULL-Preis wird durch NOT NULL Constraint abgelehnt
3
4  BEGIN;  -- Transaktion starten
5
6  -- Aktueller Zustand vor dem Test
7  SELECT 'Vor dem Test:' as info, id, name, price FROM products WHERE i
8
9  -- Versuch, Preis auf NULL zu setzen
10 UPDATE products SET price = NULL WHERE id = 1;
11
12 -- ❌ Erwartete Fehlermeldung: NOT NULL Constraint Violation
13 -- ❌ Diese Zeilen werden NICHT erreicht, da vorher eine Exception ge
     wird
14
15 ROLLBACK;  -- Wird nur bei manuellem Aufruf erreicht
16
17 -- Prüfen, dass Daten unverändert sind
```

```
SELECT 'Nach dem fehlgeschlagenen Update:' as info, id, name, price
FROM products WHERE id = 1;

-- ✅ Test erfolgreich wenn: Update wird verhindert UND Preis bleibt unverändert
```

```
-- Test 3: Edge Cases
-- Erwartung: NULL-Preis wird durch NOT NULL Constraint abgelehnt

BEGIN
```

```
current transaction is aborted, commands ignored until end of transaction block
```

Test-Zusammenfassung

Was haben wir getestet?

Test-Ergebnisse:

Test	Ziel	Erwartetes Ergebnis	Status
Test 1	Timestamp-Update	`updated_at` > `created_at`	✅ Erfolgreich
Test 2	Negative Preise	Exception wird geworfen	✅ Erfolgreich
Test 3	NULL-Werte	NOT NULL Constraint greift	✅ Erfolgreich

Best Practice:

- ✅ Schreibe Test-Scripts für jeden Trigger
- ✅ Teste Edge Cases (NULL, 0, negative Werte)
- ✅ Teste sowohl Erfolgs- als auch Fehlfälle
- ✅ **Nutze Transaktionen (BEGIN/ROLLBACK) für isolierte Tests** – so bleiben Testdaten sauber!
- ✅ Teste Performance mit vielen Zeilen (hier nicht gezeigt)

Vorteile von Transaktionen beim Testen:

- 🔄 Tests sind **wiederholbar** – keine Datenverunreinigung
- 🔒 Tests sind **isoliert** – beeinflussen sich nicht gegenseitig
- ⚡ Tests sind **schnell** – ROLLBACK ist schneller als DELETE
- ✅ Originalzustand bleibt **erhalten** – Setup muss nicht wiederholt werden

Erweiterte Tests (Optional):

- Test mit 0 als Preis (sollte erlaubt sein)
- Test mit sehr großen Zahlen
- Test mit vielen gleichzeitigen Updates
- Performance-Test mit BULK Inserts

Zusammenfassung

Was haben wir heute gelernt? Functions und Trigger sind mächtige Werkzeuge für server-seitige Logik in der Datenbank.

Kernpunkte: Functions

1. **Stored Functions** = Wiederverwendbare Logik in der Datenbank
2. **Syntax:** `CREATE FUNCTION name(params) RETURNS type AS $$ ... $$ LANGUAGE plpgsql;`
3. **Kontrollstrukturen:** `IF...THEN...ELSE` und `CASE`
4. **Fehlerbehandlung:** `RAISE EXCEPTION`
5. **Use Cases:** Berechnungen, Validierung, String-Verarbeitung

Kernpunkte: Trigger

6. **Trigger** = Automatische Reaktion auf Datenbankänderungen
7. **Trigger-Functions:** `RETURNS TRIGGER`, nutzen `OLD` und `NEW`
8. **Syntax:** `CREATE TRIGGER name BEFORE/AFTER event ON table FOR EACH ROW EXECUTE FUNCTION func();`
9. **Use Cases:** Timestamps, Audit-Logging, Validierung, Soft Delete
10. **Gefahren:** Kaskaden, Performance, Debugging-Schwierigkeiten

Wann was nutzen?

Szenario	Lösung
Einfache Validierung	✅ CHECK Constraint
Default-Werte	✅ DEFAULT Clause
Automatische Timestamps	✅ Trigger (UPDATE) + DEFAULT (INSERT)
Audit-Logging	✅ Trigger
Soft Delete	✅ Trigger oder App-Logik
Komplexe Berechnungen	✅ Function
Referentielle Integrität	✅ FOREIGN KEY

Sie haben heute 10 interaktive Demos durchgearbeitet – von einfachen Functions bis zu komplexen Triggern. Experimentieren Sie weiter! Ändern Sie die Beispiele, brechen Sie sie, fixen Sie sie wieder. So lernt man am besten!

Referenzen & Quellen

Offizielle Dokumentation

- PostgreSQL: PL/pgSQL Functions
- PostgreSQL: Trigger Functions
- PostgreSQL: CREATE TRIGGER
- PGlite: Browser PostgreSQL

Bücher & Tutorials

- „PostgreSQL: Up and Running“ – Regina Obe & Leo Hsu (Kapitel zu Functions & Trigger)
- „Mastering PostgreSQL“ – Hans-Jürgen Schönig
- PostGIS Tutorial: Custom Functions

Best Practices

- Use the Index, Luke: Triggers & Performance
- PostgreSQL Wiki: Trigger Best Practices

Tools

- pgAdmin – Trigger-Debugging
- DBeaver – Cross-Platform Database Tool
- PGlite – PostgreSQL im Browser