

TESTY

1.Co testuje: Ustawienie kontekstu app.user, z którego korzystają triggery audytu i walidacji.

Polecenie: `SELECT set_config('app.user','admin', true);`

Oczekiwany wynik: zwróci admin.

```
[postgres=> SELECT set_config('app.user','admin', true);
set_config
-----
admin
(1 row)
```

2. Co testuje: Czy wszystkie wymagane triggery są aktywne na tabelach w schematach users/files/jobs/server.

```
postgres=> SELECT
postgres->   n.nspname AS schema,
postgres->   c.relname AS table_name,
postgres->   t.tgname AS trigger_name
postgres-> FROM pg_trigger t
postgres-> JOIN pg_class c ON c.oid = t.tgrelid
postgres-> JOIN pg_namespace n ON n.oid = c.relnamespace
postgres-> WHERE NOT t.tgisinternal
postgres->   AND n.nspname IN ('users','files','jobs','server')
[postgres-> ORDER BY 1,2,3;
schema |    table_name    |          trigger_name
-----+-----+-----+
files | documentperms | trg_audit_files_documentperms
files | documentperms | trg_files_validate_documentperms
files | documents     | trg_audit_files_documents
files | documents     | trg_files_auto_owner_perms
jobs  | jobdict       | trg_audit_jobs_jobdict
jobs  | jobdict       | trg_jobs_create_counters
jobs  | jobtranslations | trg_audit_jobs_jobtranslations
jobs  | jobtranslations | trg_jobs_translation_fallback
users | user          | trg_audit_users_user
users | user          | trg_users_block_login_change
users | user          | trg_users_hash_password
(11 rows)

postgres=>
```

Oczekiwany wynik: Lista ok. 11 triggerów, m.in.:

- `trg_users_hash_password`, `trg_users_block_login_change`
- `trg_files_auto_owner_perms`, `trg_files_validate_documentperms`
- `trg_jobs_create_counters`, `trg_jobs_translation_fallback`
- `trg_audit_*` dla users/files/jobs.

3. Co testuje:

- Funkcję users.register_user
- Trigger trg_users_hash_password (hash i salt)

```
postgres=> SELECT users.register_user(
postgres(>   'doc_user_1', 'Pass123!', 22, 'AGH', 'Polish', false, false,
postgres(>   'STUDENT', 'EN', 'DARK'
postgres(> );
register_user
-----
(1 row)
```

Oczekiwany wynik: 1 wiersz zwrócony (void), bez błędu.

4. Co testuje: Czy hasło nie jest przechowywane jako plaintext i czy salt jest ustawiony.

```
postgres=> SELECT login, password, salt
postgres-> FROM users."user"
postgres-> WHERE login='doc_user_1';
      login      |          password          |           salt
-----+-----+-----+
 doc_user_1 | 48484830807207b1c5e3595dcc922953 | 3d9fbb93aaec07896ab30f7f0425b8e9
(1 row)
```

Oczekiwany wynik: password = długi hash (nie Pass123!), salt ≠ NULL

5. Co testuje: Funkcję users.check_login dla poprawnego hasła.

```
[postgres=> SELECT users.check_login('doc_user_1','Pass123!') AS should_be_true;
should_be_true
-----
t
(1 row)
```

Oczekiwany wynik: t

6. Co testuje: Funkcję users.check_login dla złego hasła.

```
[postgres=> SELECT users.check_login('doc_user_1','wrong') AS should_be_false;
should_be_false
-----
f
(1 row)
```

7. Co testuje: Trigger trg_users_block_login_change blokujący zmianę loginu.

```
postgres=> DO $$  
postgres$> BEGIN  
postgres$>   UPDATE users."user" SET login='doc_user_1_hacked' WHERE login='doc_user_1';  
postgres$>   RAISE EXCEPTION 'FAIL: login change should be blocked';  
postgres$> EXCEPTION WHEN others THEN  
postgres$>   RAISE NOTICE 'OK expected: %', SQLERRM;  
postgres$> END $$;  
NOTICE:  OK expected: Changing login is not allowed  
DO
```

Oczekiwany wynik: NOTICE z komunikatem w stylu: Changing login is not allowed.

8. Co testuje: Czy trigger hashujący działa też przy zmianie hasła.

```
postgres=> UPDATE users."user"  
postgres-> SET password='NewPass!234'  
postgres-> WHERE login='doc_user_1';  
UPDATE 1  
postgres=>  
[postgres=> SELECT users.check_login('doc_user_1','NewPass!234') AS should_be_true_after_update;  
should_be_true_after_update  
-----  
t  
(1 row)
```

Oczekiwany wynik: UPDATE 1 + t.

Files: documents + permissions + grant/revoke + validate

9. Co testuje: Ustawienie kontekstu app.user dla operacji na permsach.

```
[postgres=> SELECT set_config('app.user','admin', true);  
set_config  
-----  
admin  
(1 row)
```

Oczekiwany wynik: admin

10. Co testuje: Czy tabela files.permissions jest wypełniona i jakie akcje odpowiadają poziomom.

```

postgres=> SELECT * FROM files.permissions ORDER BY permissionlevel;
   permissionlevel | read_perm | delete_perm | share_perm
   +-----+-----+-----+
   1 | t | f | f
   2 | t | t | f
   3 | t | t | t
(3 rows)

```

Oczekiwany wynik: 3 wiersze (1/2/3) z flagami read_perm/delete_perm/share_perm.

11. Co testuje:

- INSERT do files.documents
- trigger trg_files_auto_owner_perms (auto dopisanie ownera do documentperms)

```

postgres=> INSERT INTO files.documents(did, owner, title, path)
postgres-> VALUES (101, 'alice', 'CV Doc Test', 'azure.database.com/files/cv_doc_test.pdf');
INSERT 0 1

```

Oczekiwany wynik: INSERT 0 1

12. Co testuje: Czy trigger dopisał ownera z maksymalnym permissionlevel (zwykle 3).

```

postgres=> SELECT * FROM files.documentperms
postgres-> WHERE did = 101
postgres-> ORDER BY pid;
   pid | did | login | permissionlevel
   +-----+-----+-----+
   3 | 101 | alice | 3
(1 row)

```

Oczekiwany wynik: 1 wiersz dla owner z permissionlevel = 3.

13. Co testuje: Czy owner ma prawo do read/share/delete.

```

postgres=> SELECT
postgres->   files.has_permission('alice', 101, 'read') AS owner_read,
postgres->   files.has_permission('alice', 101, 'share') AS owner_share,
postgres->   files.has_permission('alice', 101, 'delete') AS owner_delete
   owner_read | owner_share | owner_delete
   +-----+-----+-----+
   t | t | t
(1 row)

```

Oczekiwany wynik: same t.

14. Co testuje: Przygotowanie konta do testów grant/revoke.

```
postgres=> SELECT users.register_user(
postgres(>   'bob_doc_1', 'Secret#1', 30, NULL, 'Czech', true, false,
postgres(>   'COMPANY', 'EN', 'LIGHT'
[postgres(> );
register_user
-----
(1 row)
```

Oczekiwany wynik: bez błędu.

15. Co testuje: Brak uprawnień dopóki owner nie nada.

```
[postgres=> SELECT files.has_permission('bob_doc_1', 101, 'read') AS bob_read_before;
bob_read_before
-----
f
(1 row)
```

Oczekiwany wynik: f

16. Co testuje: Funkcję files.grant_permission i zapis do documentperms.

```
postgres=> SELECT files.grant_permission('alice', 101, 'bob_doc_1', 1);
grant_permission
-----
(1 row)

postgres=>
postgres=> SELECT * FROM files.documentperms
postgres=> WHERE did = 101
[postgres=> ORDER BY pid;
pid | did | login    | permissionlevel
-----+-----+-----+
 3 | 101 | alice      |          3
 4 | 101 | bob_doc_1 |          1
(2 rows)
```

Oczekiwany wynik:

- grant bez błędu
- pojawia się wiersz dla bob_doc_1 z permissionlevel=1

17. Co testuje: Poprawne mapowanie level=1 → tylko read.

```
postgres=> SELECT
postgres->   files.has_permission('bob_doc_1', 101, 'read') AS bob_read_after,
postgres->   files.has_permission('bob_doc_1', 101, 'share') AS bob_share_after;
bob_read_after | bob_share_after
-----+-----
t      | f
(1 row)
```

Oczekiwany wynik: t i f

18. Co testuje: Blokadę w files.grant_permission (grantor musi mieć share).

```
postgres=> DO $$  
postgres$> BEGIN  
postgres$>   PERFORM files.grant_permission('bob_doc_1', 101, 'bob_doc_1', 1);  
postgres$>   RAISE EXCEPTION 'FAIL: should require share permission';  
postgres$> EXCEPTION WHEN others THEN  
postgres$>   RAISE NOTICE 'OK expected: %', SQLERRM;  
[postgres$> END $$;  
NOTICE: OK expected: User bob_doc_1 has no share permission for document 101  
DO
```

Oczekiwany wynik: NOTICE z błędem typu “has no share permission”.

19. Co testuje:

- eskalację perms do 3
- to, że user z share może używać grant_permission

```
postgres=> SELECT files.grant_permission('alice', 101, 'bob_doc_1', 3);
grant_permission
-----
(1 row)

postgres=>
postgres=> -- bob teraz daje read ownerowi (albo komuś innemu istniejącemu)
postgres=> SELECT files.grant_permission('bob_doc_1', 101, 'alice', 1);
grant_permission
-----
(1 row)

postgres=>
postgres=> SELECT * FROM files.documentperms
postgres=> WHERE did = 101
postgres=> ORDER BY pid;
 pid | did | login    | permissionlevel
----+---+-----+
  3 | 101 | alice    |                  1
  4 | 101 | bob_doc_1 |                  3
(2 rows)
```

Oczekiwany wynik: brak błędów, bob ma 3.

20. Co testuje: Działanie triggera trg_files_validate_documentperms przy bezpośrednim INSERT.

```

postgres=> SELECT users.register_user(
postgres(>   'charlie_doc_3', 'Char#123', 25, NULL, 'Polish', false, false,
postgres(>   'STUDENT', 'EN', 'DARK'
postgres(> );
register_user
-----
(1 row)

postgres=>
postgres=> -- aktor = bob (ma share=3)
postgres=> SELECT set_config('app.user','bob_doc_1', true);
set_config
-----
bob_doc_1
(1 row)

postgres=>
postgres=> INSERT INTO files.documentperms(pid, did, login, permissionlevel)
postgres=> VALUES (12001, 101, 'charlie_doc_1', 3);
ERROR: duplicate key value violates unique constraint "documentperms_pkey"
DETAIL: Key (pid)=(12001) already exists.
postgres=>
postgres=> SELECT * FROM files.documentperms
postgres=> WHERE did = 101
postgres=> ORDER BY pid;
 pid | did |      login       | permissionlevel
-----+-----+-----+-----+
  3 | 101 |    alice     |          1
  4 | 101 | bob_doc_1 |          3
12001 | 101 | charlie_doc_1 |          3
(3 rows)

```

Oczekiwany wynik:

W Twojej aktualnej implementacji (jak u Ciebie wyszło wcześniej): **INSERT przejdzie**, bo bob ma share i level=3.

W dokumentacji opisać to jako: “User z share może bezpośrednio modyfikować documentperms (do swojego poziomu).”

21. Co testuje: Uporządkowanie danych po dokumentacji.

```

postgres=> DELETE FROM files.documentperms
postgres=> WHERE did = 101 AND pid = 12001;
DELETE 1
postgres=>

```

Oczekiwany wynik: rekord znika.

Jobs: jobdict → counters + bump_counter + translation fallback

22. Co testuje: Ustawienie kontekstu app.user (żeby audit wiedział kto robi zmiany).

```
[postgres=> SELECT set_config('app.user','admin', true);
  set_config
-----
  admin
(1 row)
```

Oczekiwany wynik: admin

23. Co testuje: INSERT do jobs.jobdict + uruchomienie triggera trg_jobs_create_counters.

```
postgres=> INSERT INTO jobs.jobdict(title_en, skill_en)
postgres-> VALUES ('Data Engineer', 'SQL')
[postgres-> RETURNING jobid;
  jobid
-----
  344249
(1 row)

INSERT 0 1
```

Oczekiwany wynik: dostaniesz nowy jobid (np. 400001). U nas 344249

24. Co testuje: Trigger fn_create_job_counters utworzył wiersze w obu tabelach liczników.

```
postgres=> SELECT * FROM jobs.jobspl WHERE jobid = 344249;
  jobid | countt0 | countt1 | countt2
-----+-----+-----+
  344249 |      0 |      0 |      0
(1 row)

postgres=> SELECT * FROM jobs.jobscz WHERE jobid = 344249;
  jobid | countt0 | countt1 | countt2
-----+-----+-----+
  344249 |      0 |      0 |      0
(1 row)
```

Oczekiwany wynik: po 1 wierszu, wszystkie liczniki na start = 0.

25. Co testuje: Funkcję jobs.bump_counter oraz poprawne mapowanie bucketów T0/T1/T2.

```
postgres=> SELECT jobs.bump_counter('pl', 344249, 'T2');
      bump_counter
-----
(1 row)

postgres=> SELECT jobs.bump_counter('cz', 344249, 'T1');
      bump_counter
-----
(1 row)

postgres=>
postgres=> SELECT * FROM jobs.jobspl WHERE jobid = 344249;
   jobid | countt0 | countt1 | countt2
-----+-----+-----+
  344249 |      0 |      0 |      1
(1 row)

postgres=> SELECT * FROM jobs.jobscz WHERE jobid = 344249;
   jobid | countt0 | countt1 | countt2
-----+-----+-----+
  344249 |      0 |      1 |      0
(1 row)
```

Oczekiwany wynik:

- jobspl.countt2 zwiększy się o 1
- jobscz.countt1 zwiększy się o 1

26. Co testuje: Obsługę błędu “unknown country”.

```
postgres=> DO $$
postgres$> BEGIN
postgres$>   PERFORM jobs.bump_counter('xx', 344249, 'T1');
postgres$>   RAISE EXCEPTION 'FAIL: unknown country should fail';
postgres$> EXCEPTION WHEN others THEN
postgres$>   RAISE NOTICE 'OK expected: %', SQLERRM;
postgres$> END $$;
NOTICE:  OK expected: Unknown country: xx (use pl/cz)
DO
```

Oczekiwany wynik: NOTICE z komunikatem typu: Unknown country: xx (use pl/cz).

27. Co testuje: Obsługę błędu “bucket must be T0, T1 or T2”.

```
postgres=> DO $$  
postgres$> BEGIN  
postgres$>   PERFORM jobs.bump_counter('pl', 344249, 'BAD');  
postgres$>   RAISE EXCEPTION 'FAIL: BAD bucket should fail';  
postgres$> EXCEPTION WHEN others THEN  
postgres$>   RAISE NOTICE 'OK expected: %', SQLERRM;  
[postgres$> END $$;  
NOTICE: OK expected: bucket must be T0, T1 or T2  
DO
```

Oczekiwany wynik: NOTICE z komunikatem: bucket must be T0, T1 or T2.

28. Co testuje: Trigger trg_jobs_translationFallback (fallback do title_en/skill_en).

```
postgres=> INSERT INTO jobs.jobtranslations(jobid, language, local_title, local_skill)  
postgres-> VALUES (344249, 'EN', NULL, NULL)  
postgres-> ON CONFLICT (jobid, language) DO UPDATE  
postgres-> SET local_title = EXCLUDED.local_title,  
postgres->     local_skill = EXCLUDED.local_skill;  
INSERT 0 1  
postgres=>  
postgres=> SELECT * FROM jobs.jobtranslations  
postgres-> WHERE jobid = 344249 AND language = 'EN';  
jobid | language | local_title | local_skill  
-----+-----+-----+-----  
344249 | EN      | Data Engineer | SQL  
(1 row)
```

Oczekiwany wynik:

local_title = 'Data Engineer' oraz local_skill = 'SQL'.

29. Co testuje: Że trigger nie niszczy danych, gdy user poda local_title/local_skill.

```
postgres=> UPDATE jobs.jobtranslations  
postgres-> SET local_title = 'Inżynier Danych', local_skill = 'SQL / ETL'  
postgres-> WHERE jobid = 344249 AND language = 'EN';  
UPDATE 1  
postgres=>  
postgres=> SELECT * FROM jobs.jobtranslations  
postgres-> WHERE jobid = 344249 AND language = 'EN';  
jobid | language | local_title | local_skill  
-----+-----+-----+-----  
344249 | EN      | Inżynier Danych | SQL / ETL  
(1 row)
```

Oczekiwany wynik: Widzisz dokładnie Twoje wartości, bez “powrotu” do angielskich.

Audit + Server

30. Co testuje: Ustawienie app.user używane przez trigger audytu.

```
[postgres=> SELECT set_config('app.user','admin', true);
set_config
-----
admin
(1 row)
```

To jest 3 razy to samo na początku każdej sekcji

Oczekiwany wynik: admin

31. Co testuje: Funkcję server.is_ip_allowed(ip, scope) i dane w server.ipwhitelist.

```
postgres=> SELECT * FROM server.ipwhitelist ORDER BY ip;
      ip      | canconsole | candb | canclient
-----+-----+-----+
 127.0.0.1 | t          | t     | t
(1 row)

postgres=>
postgres=> SELECT server.is_ip_allowed('127.0.0.1','console') AS console_ok;
  console_ok
-----
 t
(1 row)

postgres=> SELECT server.is_ip_allowed('127.0.0.1','db')      AS db_ok;
  db_ok
-----
 t
(1 row)

postgres=> SELECT server.is_ip_allowed('127.0.0.1','client') AS client_ok;
  client_ok
-----
 t
(1 row)
```

Oczekiwany wynik:

- w ipwhitelist widać rekordy (np. 127.0.0.1)
- is_ip_allowed zwraca t dla scope, które masz ustawione na true (console/db/client)

32. Co testuje: Czy triggery audytu zapisują DML do server.logs.

```
postgres=> -- 1) UPDATE usera (powinien dać audit UPDATE on users.user)
postgres=> UPDATE users."user"
postgres-> SET age = age
postgres-> WHERE login = 'doc_user_1';
UPDATE 1
postgres=>
postgres=> -- 2) INSERT/DELETE na jobtranslations (audit na jobs.jobtranslations)
postgres=>
postgres=> INSERT INTO jobs.jobtranslations(jobid, language, local_title, local_skill)
postgres=> VALUES (344249, 'PL', 'Tester', 'SQL')
postgres=> ON CONFLICT (jobid, language) DO UPDATE
postgres=> SET local_title = EXCLUDED.local_title,
|postgres=>     local_skill = EXCLUDED.local_skill;
INSERT 0 1
```

Oczekiwany wynik: brak błędów.

33. Co testuje: Czy server.logs rzeczywiście zbiera wpisy z triggerów audytu.

```
postgres=> SELECT *
postgres=> FROM server.logs
postgres=> ORDER BY date DESC, "user";
      date   |    user    |          message          |      ip       | type
-----+-----+-----+-----+-----+
2026-01-24 | azureuser | INSERT on users.user; INSERT on users.user; INSERT on files.documents; INSERT on files.documentperms; INSERT on jobs.jobdict; INSERT on users.user; UPDATE on users.user; INSERT on jobs.jobtranslations; INSERT on users.user; INSERT on files.documentperms; | 91.150.222.250 | INSERT
(1 row)
```

Oczekiwany wynik:

Widzisz rekord dla admin z message, w którym są wpisy typu:

- UPDATE on users.user
- INSERT/UPDATE on jobs.jobtranslations
- wcześniejsze: INSERT on files.documents, INSERT on files.documentperms, INSERT on jobs.jobdict, itd.

34. Co testuje: Czy audit zapisuje ip w server.logs.

```
postgres=> SELECT date, "user", ip, message
postgres=> FROM server.logs
postgres=> ORDER BY date DESC, "user";
      date   |    user    |      ip       |          message
-----+-----+-----+-----+
2026-01-24 | azureuser | 91.150.222.250 | INSERT on users.user; INSERT on users.user; INSERT on files.documents; INSERT on files.documentperms; INSERT on jobs.jobdict; INSERT on users.user; UPDATE on users.user; INSERT on jobs.jobtranslations; INSERT on users.user; INSERT on files.documentperms;
(1 row)
```

Oczekiwany wynik: kolumna ip ma wartość

35. Co testuje: Czy audyt jest podpięty do wymaganych tabel (users.user, jobs.* , files.*).

```
postgres=> SELECT
postgres->   n.nspname AS schema,
postgres->   c.relname AS table_name,
postgres->   t.tgname AS trigger_name
postgres-> FROM pg_trigger t
postgres-> JOIN pg_class c ON c.oid = t.tgrelid
postgres-> JOIN pg_namespace n ON n.oid = c.relnamespace
postgres-> WHERE NOT t.tgisinternal
postgres->   AND t.tgname LIKE 'trg_audit_%'
|postgres-> ORDER BY 1,2,3;
schema | table_name | trigger_name
-----+-----+-----
files  | documentperms | trg_audit_files_documentperms
files  | documents    | trg_audit_files_documents
jobs   | jobdict      | trg_audit_jobs_jobdict
jobs   | jobtranslations | trg_audit_jobs_jobtranslations
users  | user          | trg_audit_users_user
(5 rows)
```

Oczekiwany wynik: kilka triggerów trg_audit_* na tabelach users/files/jobs.