Lancer les tests E2E (Playwright)

Cette page décrit comment démarrer l'environnement local et lancer les tests d'intégration (Playwright, dossier `tests/` du dépôt *logeas-web*).

Pourquoi un environnement 100% local

Les tests E2E doivent toujours tourner contre des serveurs 100% locaux, jamais contre les serveurs en ligne. Ce ne sont pas des tests en lecture seule : ils créent et effacent de vraies bases, tickets, sessions de formation, etc.

- `src/environments/environment.ts` est basculé à la main entre local et en ligne selon les

besoins de debug du moment — son état à un instant donné n'est **jamais fiable**.

- L'appli Angular des tests doit donc être démarrée avec `npm run start:local` (pas `npm start`), qui utilise la configuration Angular `local` (voir `angular.json`) pointant sur

`environment.local.ts` — un fichier séparé, jamais basculé à la main, avec des URL

```
  ''localhost'' figées.
```

1. Démarrer les serveurs locaux

Trois serveurs backend (PGI, LoGeAsWebServeur, Nono) plus le serveur Angular sont nécessaires. Les trois backends doivent tourner **en administrateur**, sinon la création de base échoue.

Un script fait tout en une seule confirmation UAC :

```
powershell -ExecutionPolicy Bypass -File scripts\demarrer-serveurs-
locaux.ps1
```

Ce script :

- démarre PGI (port 8084), LoGeAsWebServeur (port 8086), NonoServeur (port 8089) puis Angular

(`npm run start:local`, port 4200) ;

- ignore un serveur déjà démarré (pas de fenêtre en double si on le relance) ;
- efface `generic.s3db` et `tickets.s3db` dans le dossier de NonoServeur avant de le

démarrer (uniquement s'il n'est pas déjà lancé) : ces fichiers accumulent les tickets et

```
  sessions de formation créés par les runs E2E successifs sans jamais être
nettoyés, ce qui finit
```

```
  par ralentir les tests.
```

Une fois le script terminé, les 4 serveurs tournent chacun dans leur propre fenêtre console.

2. Lancer les tests

Toute la suite

```
npm run test-integration
```

C'est la commande utilisée en CI. Elle exécute la liste de specs définie dans le script `test-integration` de `package.json`, dans l'ordre (`verification-sauvegarde.spec.ts` et `verification-archive-fiscale.spec.ts` en dernier, pour accumuler un maximum de données réelles avant de les sauvegarder/archiver).

Un seul fichier

```
npx playwright test <fichier>.spec.ts --no-deps
```

Exemple :

```
npx playwright test fiche-intervention.spec.ts --no-deps
```

Le –`no-deps` est important pour certains specs (`verification-sauvegarde.spec.ts`, `verification-archive-fiscale.spec.ts`) : sans lui, Playwright rejoue toute la suite `chromium` avant, à cause des dépendances de projet définies dans `playwright.config.ts`.

Réutiliser la base d'un run précédent

Chaque run recrée normalement une base de test vide (jusqu'à ~20 min). Pour itérer plus vite sur un seul spec sans repasser par cette création :

```
PW_REUSE_BASE=1 npx playwright test <fichier>.spec.ts --no-deps
```

La base du run précédent n'est effacée qu'au **début** du run suivant (pas à la fin), donc elle est encore disponible juste après. ⚠ Si un autre processus a modifié la base entre-temps (ex. reset manuel), ce flag peut échouer avec “Base introuvable” — relancer sans PW_REUSE_BASE dans ce cas.

Voir le rapport

```
npx playwright show-report
```

3. Identifiants et fichiers locaux

Tous les identifiants/état locaux vivent dans `tests/.local/` (dossier ignoré par Git) :

Fichier	Contenu
`credentials-admin.json`	Compte PROPRIÉTAIRE (créateur/possesseur de la base de test)
`credentials.json`	Compte CIBLE (celui que les scénarios ajoutent/retirent des bases)
`credentials-mail.json`	Identifiants IMAP de la boîte de test (vérification de réception de mail réelle)
`test-base.json`	SUID/titre de la base créée par le run en cours
`storageState.json`	Session navigateur réutilisée entre les specs d'un même run

S'ils n'existent pas, `global-setup.ts` les demande de façon interactive au premier run (et propose de les sauvegarder localement pour les fois suivantes).

4. Écrire un nouveau test

1. Créer `tests/<nom>.spec.ts`, sur le modèle d'un spec existant proche (ex. `creation-ticket-client.spec.ts`).
1. Si le scénario nécessite une action serveur “hors écran” (ex. simuler une action assistance sans vrai compte assistance), ajouter une action à

```
  ''src/app/Autre-Fonction-Annexe/test-creation-base-vide/test-creation-
base-vide.ts'' plutôt que
  de piloter un écran réel — voir les actions existantes
(''creerTicketAppel'',
  ''simulerFicheIntervention'', etc.) comme modèle.
- Lier le test à la cartographie : poser l'annotation ''cartoTestId'' (voir
''tests/helpers/
  carto-logger.ts'') avec l'ID du Test correspondant côté écran "Univers
Tests > Suivi des
  tests".
- Ajouter le fichier à la liste de specs du script ''test-integration'' dans
''package.json''
  pour qu'il tourne en CI.
```

Pièges connus

- **CRUD('Read', …) avec un filtre côté serveur Nono (`ticket/CRUD`) ignore ce filtre** et renvoie toute la table — filtrer côté client (JS) après lecture complète.
- **DevExtreme et les événements de saisie** : un `dx-text-box` sans `valueChangeEvent=“keyup”`

ne synchronise sa valeur Angular qu'au blur/Entrée, pas à chaque frappe — `.fill()` seul ne

```
  suffit pas, utiliser ''.pressSequentially()'' puis ''.press('Enter')'' ou
cliquer ailleurs.
* **Élévation** : ce script ne peut pas être fermé/relancé depuis une
session sans les mêmes
  droits (erreur "Accès refusé") — redémarrer les serveurs à la main si
besoin (fermer la
  fenêtre console concernée, relancer le script).
```