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
```

Le rapport HTML Playwright (résultats détaillés, captures, traces) est régénéré à **chaque** run, directement dans `../logeas-cartographie/src/assets/playwright-report/` (voir

`playwright.config.ts`, option `reporter`), plutôt que dans le dossier par défaut du dépôt — il est donc accessible sans étape de copie manuelle depuis l'écran carto “Univers Tests” (bouton “Dernier rapport Playwright”). ⚠ Il reflète uniquement le **dernier** `npx playwright test` lancé, pas un cumul entre plusieurs runs : lancer un seul fichier écrase les résultats des autres specs qui y figuraient avant.

À côté, le bouton “Résumé et analyse du dernier rapport Playwright” ouvre un document **différent** et **statique** (`logeas-cartographie/src/assets/rapport-tests-e2e-qualite.html`) : une synthèse qualité rédigée à la main (méthodologie, catalogue de défauts trouvés, fiabilité résiduelle), pas régénérée automatiquement. Voir section 5 ci-dessous pour la maintenir à jour.

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).

`global-setup.ts` fait plus que créer la base : il pilote aussi le vrai écran “Administration > Gestion des droits > Ajouter un utilisateur” pour donner au compte CIBLE un accès **réel** à la base fraîchement créée (au-delà du simple forçage de “dernière base”) — sans ça, tout scénario qui fait agir la cible elle-même via l'UI (pas seulement le compte propriétaire/admin) reste bloqué par le popup “Choix du contexte de travail” (`listeBasesAccessibles` vide côté serveur).

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`).
2. 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.
3. 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”.

1. Ajouter le fichier à la liste de specs du script `test-integration` dans `package.json` pour qu'il tourne en CI.

Pour vérifier la réception d'un mail réel (confirmation d'inscription, notification de réponse, etc.), utiliser `tests/helpers/mail-checker.ts` (`attendreMail({sujetContient, expediteurContient},depuis, timeoutMs)`), qui interroge par IMAP réel la boîte de test (identifiants dans `tests/.local/credentials-mail.json`), plutôt que de se fier à un log serveur.

5. Garder le rapport qualité E2E à jour

Le document de synthèse (`logeas-cartographie/src/assets/rapport-tests-e2e-qualite.html`) est rédigé à la main, section par section — rien ne le régénère automatiquement quand un spec est ajouté. Pour savoir s'il a pris du retard sur les fichiers réels de `tests/` :

```
npm run verifier-rapport-qualite
```

Compare la liste des `tests/*.spec.ts` réels à ce qui est référencé dans le document (section “3.1 Synthèse des scénarios”) et liste les fichiers manquants — sans rien générer automatiquement, le texte de chaque ligne (“Ce qu'il vérifie”) restant trop nuancé pour être extrait fiablement d'un commentaire de code. À lancer après tout ajout de spec, puis compléter le tableau à la main (ou via Claude) avec les lignes manquantes.

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).
* **Envoi de mail fire-and-forget**
(''ticket.service.ts::EnvoiMailContactClient'') : déclenché
  via un ''subscribe()'' interne, pas chaîné dans l'Observable attendu par
  ''setTicketAndReponse'' — naviguer la ''page'' (ex. vers le formulaire de
connexion) juste après
  avoir reçu le signal de fin d'une action ''test-creation-base-vide'' peut
tuer cet envoi encore
  en vol. Utiliser un nouvel onglet (''page.context().newPage()'') pour la
suite du scénario
  plutôt que de réutiliser ''page'' si elle vient de déclencher un envoi de
```

```
mail asynchrone (voir
  ''reponse-assistance-mail.spec.ts'').
* **Popups multi-étapes DevExtreme** (ex. assistant "Inscription à une
Formation") : un clic sur
  l'élément qui doit faire avancer une étape ne la fait pas toujours avancer
de façon fiable —
  boucler sur le clic jusqu'à ce que l'étape suivante soit visible, plutôt
qu'un simple
  ''.click()'' suivi d'une attente fixe (voir ''inscription-
formation.spec.ts'').
* **Nettoyage de données de test partagées entre comptes** : une action
headless connectée avec un
  compte A (ex. propriétaire/admin) qui nettoie un état laissé par un run
précédent doit filtrer
  sur l'objet métier concerné (ex. l'ID de session), pas sur le compte
appelant — sinon l'état
  laissé par un AUTRE compte (ex. la cible, qui agit via l'UI dans le même
scénario) n'est jamais
  nettoyé et bloque silencieusement une réexécution (constaté sur
  ''obtenirSessionFormationTest''/''inscription-formation.spec.ts'', "Déjà
inscrit." après un
  premier run pourtant réussi).
```

From:
https://wiki-logeas.fr/certif/ - **dokuwiki-certif**

Permanent link:
https://wiki-logeas.fr/certif/doku.php?id=certif:procedure:test:playwright

Last update: **2026/09/29 08:15**