Etablissement de formation

Campus La Châtaigneraie
2 rue Charles Scherer
76240 Le Mesnil-Esnard

Entreprise d'accueil

Matmut Assurances
66 rue de Sotteville
76100 Rouen

Rapport de stage

Développement web au sein de l'équipe UX (DNI)

Pierre HOULLIERE

Du 5 janvier au 13 février 2026

2SIO - Année 2025/2026

Remerciements :

- Eric Lebeau
- Robin Szylobryt
- Romain Fregard
- Sébastien Allais
- Maxence Lefevre
- Théophile Lioppé
- Grégory Rihal
- Sébastien Pannier
- Axel Duval
- Isabelle Verguet
- Pierre Milesi
- Florent Escots
- Olivier Delange

L’entreprise

Contexte

Dates du stage

Le stage a eu lieu du lundi 5 janvier 2026 au vendredi 13 février 2026.

L’entreprise

La société Mutuelle Assurance Travailleur Mutualiste (MATMUT) a été créée le 1 janvier 1969. Sa forme juridique est Société d'assurance à forme mutuelle (SGAM). Son domaine d’activité touche la santé, l’épargne, l’automobile, l’habitation, ... En 2023, elle était catégorisée Grande Entreprise. Elle possédait 5 000 à 9 999 salariés.

Les deux principaux bâtiments du siège social.

Son siège social est domicilié au 66 rue de Sotteville 76100 Rouen. Elle possède 1382 établissements dont 561 sont en activité.

Le créateur de la Matmut, Paul Bennetot, a la volonté de proposer une assurance automobile accessible et solidaire aux travailleurs. Initialement spécialisée dans le secteur auto, la mutuelle a progressivement élargi son offre à l'assurance habitation.

Au fil des décennies, elle a su se transformer en un groupe complet, intégrant la santé, la prévoyance et l'épargne, tout en restant fidèle à ses racines normandes et à son modèle social d'origine.

Service d’affectation

Le pôle informatique est regroupé dans ce qu’on appelle la Direction du Numérique et de l'Innovation (DNI). J’ai été affecté à l’équipe n°4 (distribution 4) : l’équipe UX (expérience utilisateur).

Clients

Les clients de la Matmut, que l'on appelle techniquement des sociétaires en raison du statut mutualiste de l'entreprise, sont principalement des particuliers. Historiquement créée pour les travailleurs, elle s'est largement ouverte au fil des décennies et s'adresse aujourd'hui à toutes les catégories de la population : familles, jeunes conducteurs, étudiants ou retraités. Ces clients cherchent généralement à couvrir leurs besoins quotidiens comme l'assurance automobile, l'assurance habitation, la mutuelle santé ou la protection de la famille.

Au-delà des particuliers, elle s'adresse également aux professionnels et aux entreprises. Cela comprend les artisans, les commerçants, les professions libérales ainsi que les auto-entrepreneurs. Pour ces clients, la mutuelle propose des garanties spécifiques telles que la responsabilité civile professionnelle, la protection des locaux commerciaux ou encore des solutions de prévoyance collective pour leurs salariés.

Enfin, une partie importante de la clientèle est constituée d'agents de la fonction publique (qu'ils soient de l'État, des collectivités territoriales ou du secteur hospitalier). Cette catégorie est notamment servie par une filiale spécialisée, AMF Assurances, intégrée au Groupe Matmut. Elle accompagne aussi le monde associatif, en proposant des contrats adaptés aux besoins des structures à but non lucratif. Au total, le groupe compte aujourd'hui plus de 4.6 millions de sociétaires.

Environnement technologique de l'entreprise

Langages

Nom	Description	Logo
C# (.NET)	Assure le développement d'applications back-end robustes ainsi que des services Windows.	
JavaScript TypeScript	Permet la création d'interfaces dynamiques pour les portails clients, notamment via la bibliothèque React.	
Cobol	Gère les traitements de masse et les transactions rapides.	

Serveurs

Les serveurs de l'entreprise sont répartis en deux parties :

- Les machines "open" représentent la partie distribuée et modulaire de l'entreprise. Ces serveurs fonctionnent principalement sous des systèmes d'exploitation Linux et

Windows pour héberger tous types d'applications. On y retrouve des applications web, portails clients, outils collaboratifs, ...

- Les machines "Z" (mainframe IBM) : extrêmement puissantes fournies exclusivement par IBM. Ce système est réputé pour sa robustesse inégalée et sa capacité à traiter des millions de transactions simultanément avec une sécurité maximale. On y retrouve les applications cœur métier : contrats, sinistres, ...

Outils

L'écosystème de travail centralise la gestion de projet, la documentation et le cycle de vie du code.

Nom	Description	Logo
Jira	Permet aux équipes (développeurs, testeurs, …) de suivre l'avancement des tâches, de gérer le backlog et de piloter la correction des anomalies.	
Confluence	Utilisé comme base de connaissances et wiki d'entreprise. On y centralise toute la documentation technique, les spécifications fonctionnelles, les procédures d'exploitation et les comptes-rendus de réunions pour favoriser le partage d'informations.	
Zoom	Utilisé pour les réunions d'équipe quotidiennes, les conférences à distance et le partage d'écran.	
Azure DevOps	Gère tout le cycle de vie des applications. Permet l'hébergement du code source via les dépôts Git et automatise les phases de test et de déploiement (Pipelines CI/CD)	
TFS	Le prédécesseur d'Azure DevOps, peut encore être utilisé pour la gestion du code source et le suivi de projets sur des applications patrimoniales avant leur migration complète vers des outils plus modernes.	

Organisation du projet

Mode de transmission du travail

La transmission des missions au sein de l'équipe UX s'effectue de manière fluide et directe. Contrairement à un cadre administratif rigide, le travail m'est confié par les membres de l'équipe au fur et à mesure de l'avancement de mes tâches.

Les missions me sont données oralement dès qu'une tâche précédente est finalisée.

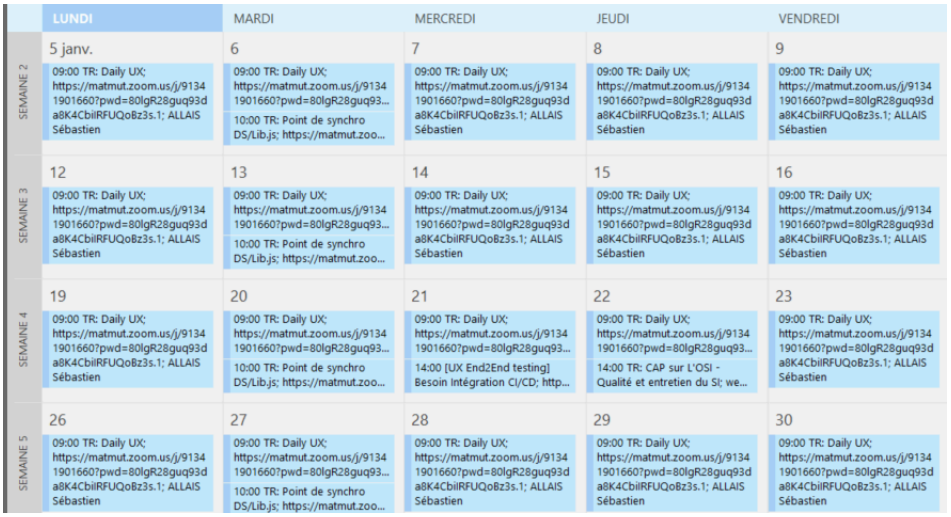
La vérification du travail s'opère par une présentation orale et technique lors du point quotidien du matin.

Pour les missions de développement, le travail est régulièrement sauvegardé et transmis via des commits sur Azure DevOps.

Outils de suivi utilisés

L'organisation de l'équipe UX ne repose pas sur une méthodologie, mais privilégie une communication quotidienne constante.

Chaque matin à 9h00, une réunion d'équipe est organisée. Elle dure généralement entre 30 minutes et 1 heure. Ce créneau permet de faire un point de situation sur l'avancement des travaux de chacun. C'est également le moment privilégié pour effectuer des démonstrations de mes missions terminées devant l'équipe.



Extrait de mon calendrier de réunions.

Ces échanges sont facilités par l'utilisation de Zoom avec le partage d'écran, permettant une coordination efficace.

Description des missions

Mise en place de l’environnement de tests end to end du design system

La Matmut s'appuie sur un design system robuste, véritable bibliothèque centralisée regroupant l'ensemble des composants graphiques (boutons, champs de formulaires, menus de navigation) utilisés de manière transverse sur tous les portails web du groupe.

Le référentiel de conception (design system).

L'enjeu est de taille : une modification mineure sur un composant de base peut provoquer un effet papillon et casser l'affichage ou les fonctionnalités de dizaines de pages différentes sans que les développeurs ne s'en aperçoivent immédiatement. L'objectif de cette mission était donc d'automatiser des tests end-to-end (e2e) pour vérifier que le design system reste parfaitement fonctionnel après chaque mise à jour.

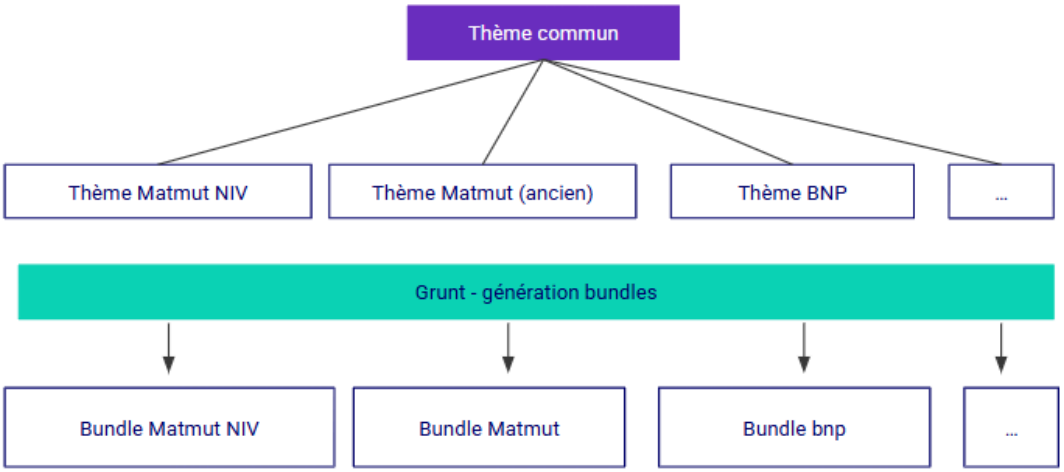
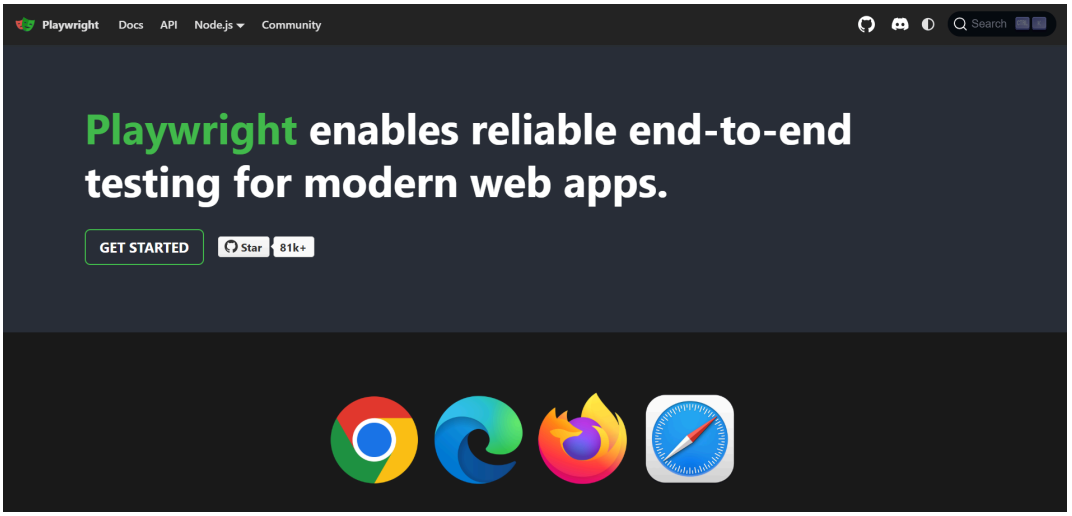


Schéma représentatif de l'architecture des données.

Ayant déjà répondu à ce genre de problématique technique lors de l'un de mes projets personnels, j'ai pris l'initiative de proposer l'outil Playwright. Ce choix s'inscrivait dans un contexte précis : l'ancien outil utilisait Robot Framework, et était trop complexe à maintenir et quasi inutilisable au quotidien.

Pour répondre aux contraintes soumises par l'équipe, il nous fallait une solution directement intégrée au design system, à la fois simple à maintenir afin de rester accessible et suffisamment performante pour une utilisation intensive. Le nouvel outil devait également être évolutif pour supporter des tests avancés, notamment la gestion d'interactions complexes comme les boutons ou les fenêtres modales.

Ce choix a été validé par l'équipe UX car Playwright permet de simuler avec une précision extrême le parcours d'un utilisateur réel (clics, saisies, navigation) dans un navigateur. Il offre même la possibilité de réaliser des tests de régression visuelle : l'outil capture une image de référence qu'il compare automatiquement à chaque nouveau lancement, garantissant ainsi une stabilité visuelle et fonctionnelle constante.



Page d'accueil de la documentation Playwright.

Pour confirmer mon choix, j'ai réalisé une analyse des différents outils disponibles en matière de tests par régression visuelle.

Solution	Usage	Stockage des images de référence	Coût	Avantage	Inconvénient
Playwright	Standalone, marche peu importe le projet.	Dans le repo.	Gratuit	• Ultra flexible. • Maintenu par Microsoft activement. • Permet de tester différents navigateurs et différents formats (mobile, desktop, ...) • S'intègre dans n'importe quel projet. • Pipeline Azure facile à mettre en place. • Image docker déjà disponible. • Peut s'intégrer dans Storybook	• Maintenance manuelle (pour changer les images de référence).
Loki	Uniquement pour Storybook.	Dans le repo.	Gratuit	• Minimaliste, mais peut être un peu trop ? • Les tests se font automatiquement, il détecte tout seul les composants.	• Uniquement des tests de VRT.
Argos CI	Standalone, marche peu importe le projet. Complémentaire à un framework de test comme Playwright ou Cypress, marche également sur Storybook avec des étapes supplémentaires.	Dans leur serveurs.	Freemium	• Pas besoin d'intégrer le SDK (Playwright, Cypress, ...) au projet, tout est géré par la plateforme. • Simple d'utilisation.	• Moins de contrôle sur les images de référence. • "Juste" un interface pour gérer les tests, alors que Playwright le fait déjà tout seul.
BackstopJS	Standalone, marche peut importe le projet.	En local.	Gratuit	• Minimaliste, mais peut être un peu trop ?	• Pas maintenu depuis plus de 2 ans. • Pas d'interface admin pour lancer/gérer les tests, uniquement via le CLI.

Tableau d'analyse réalisé sur le Confluence interne.

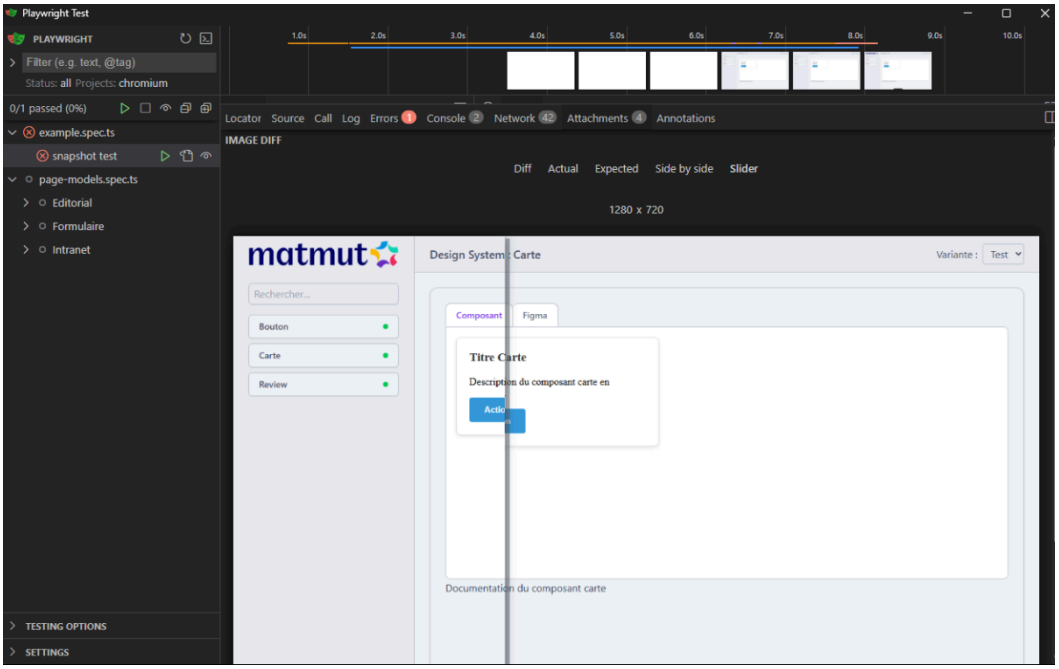
J'ai rédigé l'intégralité des tests en TypeScript. Ce choix permet de bénéficier d'un typage fort, rendant le code des tests plus robuste, plus facile à déboguer et surtout plus simple à maintenir pour les futurs développeurs de l'équipe.

J'ai structuré mes tests selon le design pattern Page Object Model. Cela permet de centraliser les sélecteurs dans des classes dédiées. Ainsi, si l'ID d'un élément change dans le HTML, je ne le modifie qu'à un seul endroit, sans avoir à réécrire tous les tests.

```
test("Hub Particulier", async ({ page }) => {
  await page.goto(getUrl("hub-particulier"));
  await expect(page).toHaveScreenshot({ fullPage: true, mask: [page.locator('.videoWrapper')] });
});
```

Exemple de rédaction d'un test

J'ai dû configurer des stratégies de masquage pour les éléments dynamiques (comme les vidéos ou les dates) afin d'éviter les flaky tests (faux positifs). Le test compare le rendu pixel par pixel avec un seuil de tolérance défini dans le fichier de configuration.

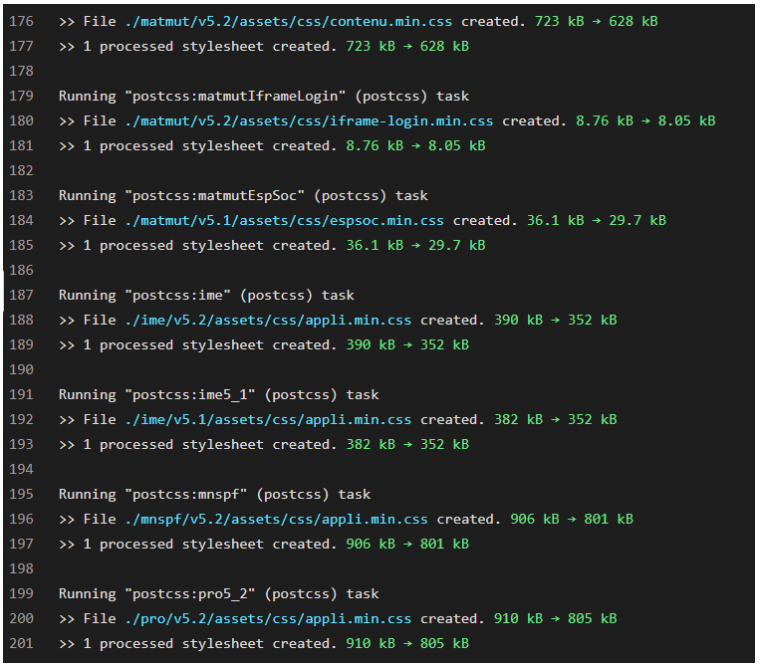


Visualisation des différences.

La partie la plus complexe de cette mission a été la réflexion autour de l'intégration continue (CI). Pour que ces tests soient réellement efficaces, ils devaient être exécutés automatiquement lors de chaque modification de code via Azure DevOps.

Développement d'une interface de monitoring pour les fichiers bundle

Au-delà de la fourniture de composants visuels, le design system a pour rôle crucial de générer des fichiers bundles (fichiers JS et CSS compressés). Ces fichiers regroupent l'ensemble du code nécessaire pour que les développeurs des différents pôles puissent les importer facilement et rapidement dans leurs propres applications.



```
>> File ./matmut/v5.2/assets/css/contenu.min.css created. 723 kB → 628 kB
>> 1 processed stylesheet created. 723 kB → 628 kB

Running "postcss:matmutIframeLogin" (postcss) task
>> File ./matmut/v5.2/assets/css/iframe-login.min.css created. 8.76 kB → 8.05 kB
>> 1 processed stylesheet created. 8.76 kB → 8.05 kB

Running "postcss:matmutEspSoc" (postcss) task
>> File ./matmut/v5.1/assets/css/espsoc.min.css created. 36.1 kB → 29.7 kB
>> 1 processed stylesheet created. 36.1 kB → 29.7 kB

Running "postcss:ime" (postcss) task
>> File ./ime/v5.2/assets/css/appli.min.css created. 390 kB → 352 kB
>> 1 processed stylesheet created. 390 kB → 352 kB

Running "postcss:ime5_1" (postcss) task
>> File ./ime/v5.1/assets/css/appli.min.css created. 382 kB → 352 kB
>> 1 processed stylesheet created. 382 kB → 352 kB

Running "postcss:mnspf" (postcss) task
>> File ./mnspf/v5.2/assets/css/appli.min.css created. 906 kB → 801 kB
>> 1 processed stylesheet created. 906 kB → 801 kB

Running "postcss:pro5_2" (postcss) task
>> File ./pro/v5.2/assets/css/appli.min.css created. 910 kB → 805 kB
>> 1 processed stylesheet created. 910 kB → 805 kB
```

Extrait des logs de la pipeline qui build les bundles.

Cependant, chaque nouvel ajout au design system alourdit ces fichiers. Si le poids du bundle devient trop important, le temps de chargement des portails pour les sociétaires s'allonge, dégradant l'expérience utilisateur. L'objectif de ma mission était de créer une interface de monitoring capable de suivre l'évolution de la taille de ces fichiers au fil des versions.

Pour répondre à cette problématique, j'ai conçu un système de collecte et d'affichage automatisé :

- J'ai modifié la pipeline Azure DevOps du projet design system en y ajoutant une étape supplémentaire. Désormais, à chaque compilation du code, un script génère automatiquement un fichier JSON. Ce fichier contient les statistiques précises du build comme la taille réelle et la taille après compression.

```
const history = config.map(theme => {
  return {
    name: theme.name,
    bundles: theme.bundles.map(bundle => {
      return {
        name: bundle.name,
        stats: {
          css: getFileSize(bundle.stats.css),
          cssGzip: getGzipSize(bundle.stats.css),
          js: getFileSize(bundle.stats.js),
          jsGzip: getGzipSize(bundle.stats.js)
        }
      };
    })
  };
});
```

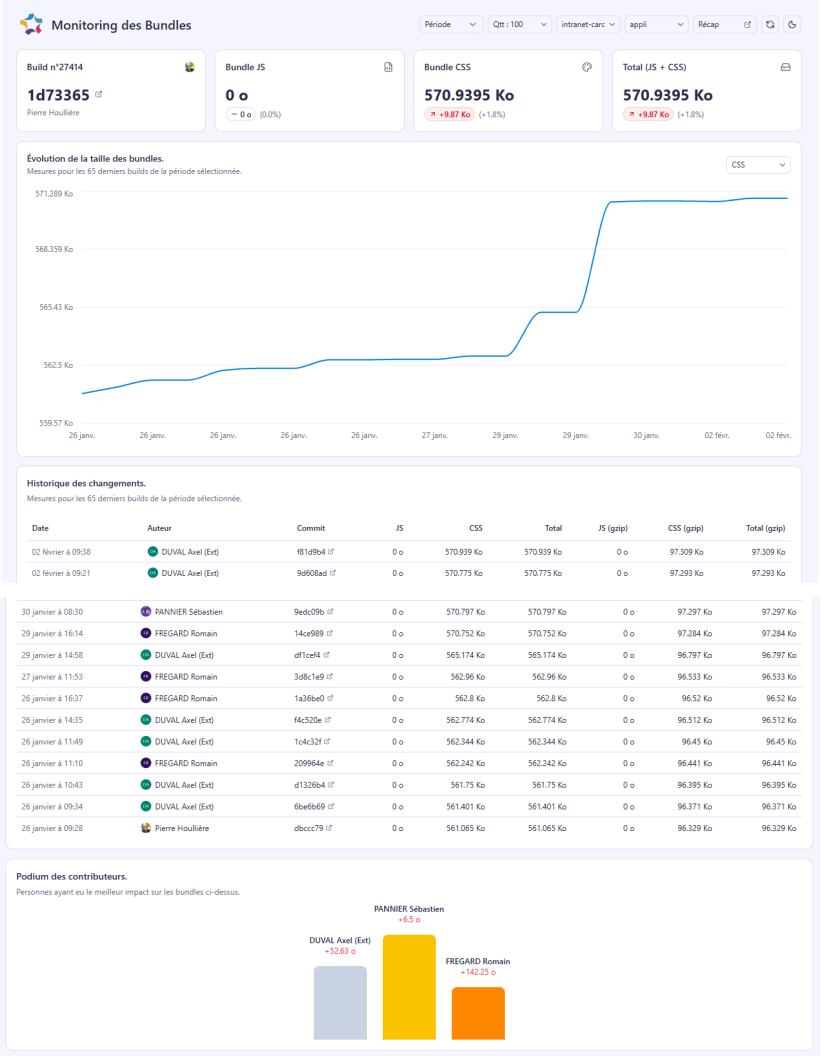
Extrait du code qui génère la structure du fichier JSON.

- Ce fichier JSON est sauvegardé en tant qu'artifact (un objet produit par le processus de build) directement sur Azure DevOps.

```yaml
- task: PublishBuildArtifacts@1
  displayName: "Publish bundles stats (JSON)"
  inputs:
    PathtoPublish: "$(Build.SourcesDirectory)/themes/monitoring-data.json"
    ArtifactName: "Stats"
    publishLocation: "Container"
```

Partie de la pipeline qui met le fichier JSON en artifact de build.

- J'ai développé une interface dédiée en React permettant de consulter ces données. Le défi technique a été de faire communiquer mon application avec l'API REST d'Azure DevOps. L'interface va chercher dynamiquement les artifacts disponibles, en extrait le contenu JSON et transforme ces données brutes en graphiques de performance lisibles par l'équipe.



Page d'accueil de l'interface de monitoring.

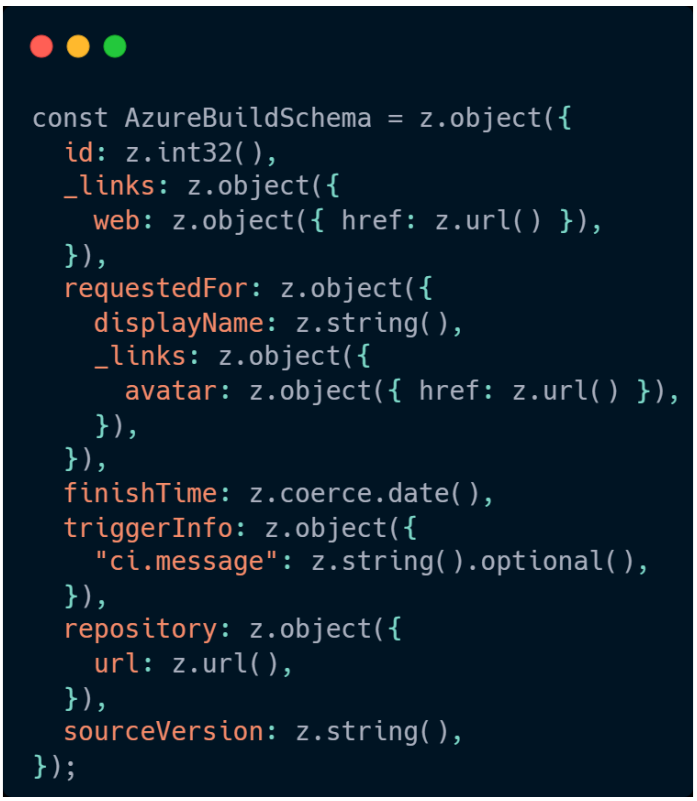
Le principal obstacle a été la mise en place de l'authentification avec l'API REST. Il a fallu configurer un token d'accès spécifique pour que mon interface puisse lire les artifacts de la pipeline.

```
class ApiService {
  private static instance: ApiService;
  private baseUrl: string;
  private authHeader: string;
  private project: string;

  private constructor() {
    this.authHeader = `Basic ${window.btoa(`:${config.pat}`)}`;
    this.baseUrl = config.org_url;
    this.project = config.project;
  }

  public static getInstance() {
    if (!ApiService.instance) {
      ApiService.instance = new ApiService();
    }

    return ApiService.instance;
  }

  private async request<T>(url: string, schema: z.ZodType<T>, responseType: "json" | "text")
{   const response = await fetch(url, { headers: { Authorization: this.authHeader } });

    if (response.status === 404) {
      console.warn(`404 error on`, url);
      return null;
    }

    if (!response.ok) {
      throw new Error(`AZDO API Error: (${response.status}) ${response.statusText}`);
    }

    let rawData: unknown;

    if (responseType === "text") {
      const text = await response.text();

      try {
        rawData = JSON.parse(text);
      } catch {
        rawData = text;
      }
    } else {
      rawData = await response.json();
    }

    const validation = schema.safeParse(rawData);

    if (!validation.success) {
      console.warn(`Validation error on ${url}`, validation.error);
      return null;
    }

    return validation.data;
  }
}
```

Extrait de la classe permettant de gérer la communication avec l'API.

Pour gérer la communication, j'ai développé une classe ApiService en implémentant le Design Pattern Singleton. Ce pattern garantit qu'une seule et unique instance de la classe existe pendant toute la durée de vie de l'application. Cela permet de centraliser la configuration de l'authentification (les headers avec le token) et d'éviter de multiplier les connexions inutiles ou les duplications de configuration à travers les composants React.

L'API d'Azure DevOps pouvant évoluer ou renvoyer des erreurs inattendues, je ne me suis pas contenté du typage statique de TypeScript. J'ai utilisé la librairie Zod pour effectuer une validation de schéma au runtime (à l'exécution).



```
const AzureBuildSchema = z.object({
  id: z.int32(),
  _links: z.object({
    web: z.object({ href: z.url() }),
  }),
  requestedFor: z.object({
    displayName: z.string(),
    _links: z.object({
      avatar: z.object({ href: z.url() }),
    }),
  }),
  finishTime: z.coerce.date(),
  triggerInfo: z.object({
    "ci.message": z.string().optional(),
  }),
  repository: z.object({
    url: z.url(),
  }),
  sourceVersion: z.string(),
});
```

Exemple de schéma Zod pour vérifier l'intégrité des données reçues.

Si le JSON renvoyé par l'API ne correspond pas strictement à la structure attendue, mon application intercepte l'erreur proprement au lieu de planter silencieusement. Cela rend l'application résiliente.

J'ai développé une couche d'abstraction générique autour de l'API fetch. La méthode est asynchrone (Promise) pour ne pas bloquer le thread principal de l'interface utilisateur (UI). J'ai également normalisé la gestion des erreurs HTTP (404, 500) en levant des exceptions typées, ce qui permet à l'interface React d'afficher des feedbacks précis à l'utilisateur en cas de problème réseau.

```
const useBuilds = (props: UseBuildsOptions) => {
  const listQuery = useQuery({
    queryKey: ["builds-list", props.from.toDateString(), props.to.toDateString(), props.quantity],
    queryFn: () => api.getBuilds(props.from, props.to, props.quantity),
    staleTime: 1000 * 30,
    placeholderData: keepPreviousData,
  });

  const buildsList = listQuery.data || [];

  const detailQueries = useQueries({
    queries: buildsList.map((build) => ({
      queryKey: ["build-artifact", build.id],
      queryFn: () =>
        limit(async () => {
          const themes = await api.getBuildArtifactFile(build.id);

          const statsBuild: StatsBuild = {
            info: build,
            themes: themes ?? [],
          };

          return statsBuild;
        }),
      staleTime: Infinity,
      gcTime: 1000 * 60 * 60 * 24 * 7,
    })),
  });

  const data = detailQueries.map((q) => q.data).filter((item): item is StatsBuild => item !== undefined);

  const isListLoading = listQuery.isLoading;
  const isDetailsLoading = buildsList.length > 0 && data.length === 0;
  const isLoading = isListLoading || isDetailsLoading;

  const isFetchingDetails = detailQueries.some((q) => q.isFetching);
  const isRefreshing = !isLoading && (listQuery.isFetching || isFetchingDetails);

  const error = listQuery.error || detailQueries.find((q) => q.error)?.error;
  const isError = listQuery.isError || detailQueries.some((q) => q.isError);

  return {
    data,
    isLoading,
    isRefreshing,
    isError,
    error,
    refetch: listQuery.refetch,
  };
};
```

Extrait du hook personnalisé permettant de récupérer les données de manière optimisée.

L'API d'Azure DevOps ne permettant pas de récupérer directement le contenu des fichiers JSON de statistiques via une simple recherche globale, j'ai dû concevoir une architecture de requêtes en cascade : il est nécessaire de récupérer d'abord la liste des builds filtrés par date, puis, pour chaque build identifié, d'effectuer une seconde requête asynchrone spécifique afin de télécharger l'artifact associé.

Le défi majeur de cette approche est le nombre exponentiel de requêtes : afficher l'historique de 50 builds déclencherait instantanément 50 requêtes HTTP simultanées, risquant de saturer le navigateur ou de provoquer une erreur côté serveur. Pour pallier ce problème, j'ai utilisé la bibliothèque p-limit. J'ai instauré une file d'attente de promesses limitée à 5 exécutions simultanées. Ainsi, même si l'application doit charger 100 fichiers, elle ne traitera que 5 requêtes en parallèle à la fois, garantissant la stabilité de l'application.

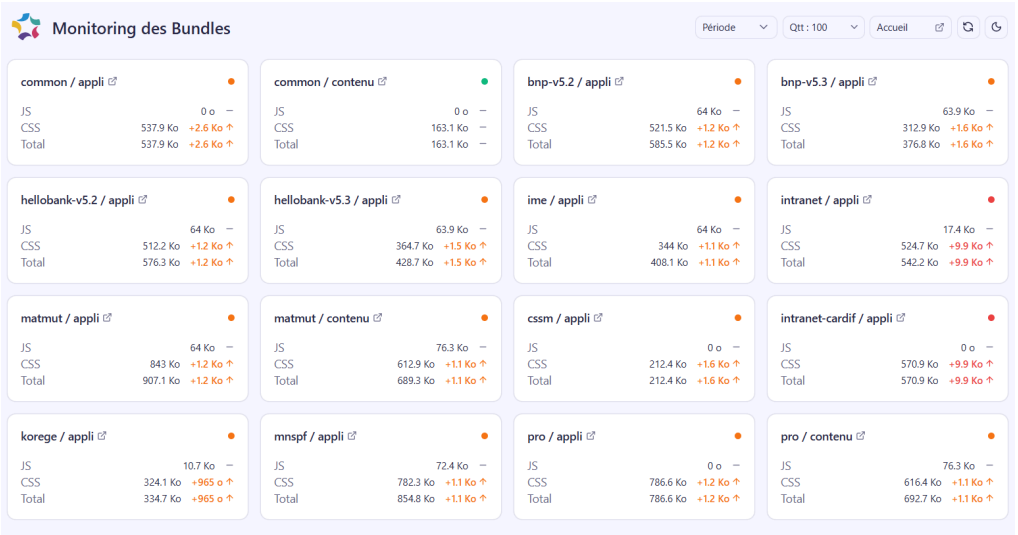
En complément de cette gestion de flux, j'ai intégré une stratégie de mise en cache persistante en utilisant IndexedDB via la bibliothèque idb-keyval. L'idée est simple : avant de solliciter le réseau, l'application vérifie si les données du build sont déjà présentes dans la

base de données locale du navigateur. Si elles s'y trouvent, elles sont récupérées instantanément, éliminant ainsi toute latence et économisant des ressources serveur. Dans le cas contraire, la requête est placée dans la file d'attente p-limit, et une fois le résultat obtenu, il est automatiquement stocké dans IndexedDB pour les consultations futures. Cette approche de mise en cache offline-first garantit une navigation fluide, même sur des historiques volumineux, tout en réduisant la charge globale sur l'API au fil des sessions.

J'ai utilisé le hook useQueries de Tanstack Query de manière dynamique. Contrairement à une requête classique, celle-ci ne s'exécute que lorsque la première requête (la liste des builds) est terminée. Le tableau de requêtes est généré à la volée (builds.map), permettant de récupérer les détails de chaque build individuellement tout en consolidant les résultats dans une seule variable d'état.

Un build passé est par définition immuable : la taille des bundles d'une version générée il y a deux semaines ne changera jamais. J'ai donc configuré une stratégie de cache spécifique pour les artifacts avec un staleTime: Infinity et un gcTime (garbage collection) de 7 jours. Concrètement, cela signifie qu'une fois les données d'un build chargées, elles ne sont plus jamais redemandées au serveur tant que l'utilisateur reste sur l'application, offrant une navigation instantanée et réduisant drastiquement la consommation de bande passante. Pour fluidifier la navigation lors du changement de filtres (dates, quantité, …), j'ai utilisé l'option placeholderData: keepPreviousData. Cela permet de conserver les données affichées à l'écran pendant le chargement des nouvelles données, évitant ainsi l'effet de scintillement désagréable d'un état de chargement intermédiaire.

Si les graphiques sont idéaux pour visualiser une tendance temporelle, l'équipe UX avait également besoin d'une vision synthétique et immédiate de l'état de santé de l'ensemble du design system. J'ai donc développé une vue qui permet de comparer, en un coup d'œil, le poids actuel de chaque bundle par rapport à une période de référence donnée. Cette vue met en exergue les variations positives ou négatives pour les bundles. J'ai donc utilisé Tanstack Router pour créer du routage dans l'application. Cela a également permis de refactoriser la logique des paramètres dans l'URL (pour stocker l'état des filtres).



Page récapitulative des évolutions des bundles.

Le traitement des données brutes impliquant des opérations coûteuses (tri de tableaux, boucles imbriquées, calculs arithmétiques), il était impératif de ne pas bloquer le thread principal à chaque rafraîchissement du composant. J'ai utilisé le hook useMemo pour mettre en cache le résultat du calcul. Ainsi, la transformation des données n'est réexécutée que si les données sources changent réellement. Cela garantit une fluidité d'interface optimale, même avec un grand volume de données.

Les données reçues de l'API sont hiérarchiques (Build -> Thèmes -> Bundles). Pour les afficher dans un tableau de données, j'ai dû écrire un algorithme d'aplatissement. Je parcours l'arbre du build le plus récent et je génère une liste linéaire d'objets normalisés. Chaque objet contient l'identifiant unique du bundle, son thème, ses valeurs actuelles et ses variations.

Le déploiement de l'application présentait un défi d'infrastructure. Créer une pipeline de déploiement dédiée au sein d'Azure DevOps aurait nécessité des délais importants de validation et de configuration serveur. Les pipelines existantes du design system étant complexes et déjà validés, j'ai opté pour une approche d'intégration embarquée plutôt que de créer un nouveau circuit de déploiement parallèle.

```
server {
    listen       8080;
    server_name  localhost;
    port_in_redirect off;
    location / {
        root   /usr/share/nginx/html;
        index  /GUI/README.md;
        add_header 'Access-Control-Allow-Origin' '*';
    }

    location /monitoring {
        alias /usr/share/nginx/html/monitoring;
        try_files $uri $uri/ /monitoring/index.html;
        add_header 'Access-Control-Allow-Origin' '*';
    }

    location = / {
        return 301 /GUI/index.html;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Configuration nginx de déploiement.

J'ai modifié le script de build pour fusionner l'application de monitoring avec le design system :

- Le projet React est compilé avec Vite pour générer des fichiers statiques optimisés (HTML, JS, CSS).

- Ces fichiers sont automatiquement déplacés dans un sous-dossier spécifique de l'arborescence du Design System.
- Lorsque la pipeline Azure DevOps génère le zip final, elle inclut désormais mon application de monitoring sans étapes supplémentaires.

Cela a permis de bénéficier immédiatement du déploiement existant sans modifier l'infrastructure.

Une fois déployée, l'application devait être accessible via une URL propre sans entrer en conflit avec les fichiers du design system. J'ai configuré un reverse proxy sur le serveur web. Ce mécanisme intercepte les requêtes ciblant l'URL /monitoring et les redirige vers le fichier index.html de mon application React. L'utilisateur accède à l'outil comme s'il faisait partie intégrante du site principal.

Lexique

Mot	Définition
API	Interface permettant à deux applications de communiquer entre elles.
Artifact	Fichier ou paquet produit automatiquement à la fin d'un processus de compilation (build), prêt à être utilisé ou déployé.
Backend	Partie immergée d'une application (côté serveur). Elle gère la logique, les calculs et l'accès aux données, par opposition au front-end.
Backlog	Liste priorisée des fonctionnalités à développer ou des bugs à corriger dans un projet agile.
Build	Processus de transformation du code source écrit par le développeur en un logiciel ou un fichier exécutable.
Bundle	Fichier unique regroupant et compressant l'ensemble des scripts (JS, CSS) d'une application pour optimiser son chargement.
CI/CD	Ensemble de pratiques automatisant les tests et la mise en production du code à chaque modification.
Design system	Bibliothèque centrale regroupant les composants graphiques (boutons, formulaires) et les règles visuelles pour assurer la cohérence des applications d'une entreprise.
Framework	Ensemble d'outils et de composants logiciels fournissant une structure de base pour développer des applications.
Frontend	Partie visible de l'application avec laquelle l'utilisateur interagit (interface)
Mainframe	Ordinateur central très puissant utilisé par les grandes entreprises pour traiter des volumes massifs de transactions avec une fiabilité extrême.
Pipeline	Suite d'étapes automatisées (compilation, tests, déploiement) exécutée

	par un outil comme Azure DevOps lors d'une modification du code.
Playwright	Outil permettant d'automatiser des tests en simulant les actions d'un véritable utilisateur sur un navigateur web.
React	Bibliothèque JavaScript populaire développée par Facebook, utilisée pour construire des interfaces utilisateurs dynamiques.
Régression visuelle	Technique de test consistant à comparer des captures d'écran (avant/après) pour détecter des modifications graphiques indésirables.
Tests e2e	Tests de bout en bout qui vérifient le bon fonctionnement d'une application dans son ensemble, du point de vue de l'utilisateur final.
TypeScript	Surcouche du langage JavaScript qui ajoute un typage strict, permettant de rendre le code plus robuste et plus facile à maintenir.
Design pattern	Solution standardisée et réutilisable pour répondre à un problème récurrent de conception logicielle.
Singleton	Patron de conception (Design Pattern) garantissant qu'une classe n'a qu'une seule instance et fournissant un point d'accès global à celle-ci.
Page Object Model	Pattern d'architecture de test qui consiste à créer une classe pour chaque page de l'application. Cela sépare la logique de test (scénario) de la structure de la page (sélecteurs HTML).
Hook	Fonction spéciale de React (commençant par use, comme useMemo ou useQuery) permettant d'utiliser l'état et d'autres fonctionnalités sans écrire de classe.
Memoisation	Technique d'optimisation (utilisée via useMemo) consistant à mettre en cache le résultat d'un calcul coûteux pour éviter de le refaire si les données d'entrée n'ont pas changé.
Immutabilité	Principe selon lequel un objet ne peut pas être modifié après sa création. En React, cela oblige à créer des copies des données (via [...data]) plutôt que de les modifier directement, assurant la fiabilité des rendus.
Rate limiting	Technique de contrôle du débit de données pour limiter le nombre de requêtes simultanées vers l'API et éviter de surcharger le serveur ou le navigateur.
Promise	Objet JavaScript représentant l'état (en cours, réussi ou échoué) d'une opération asynchrone, comme une requête réseau vers une API.
Reverse proxy	Serveur intermédiaire qui récupère les requêtes d'un client (navigateur) pour les transmettre à un ou plusieurs serveurs internes.
Vite	Outil de compilation pour les applications web. Il est beaucoup plus rapide que ses prédécesseurs (comme Webpack) et produit des builds optimisés pour la production.

Conclusion

Avis personnel

L'aspect le plus gratifiant de mon stage a été de travailler sur des outils internes. Contrairement à des projets académiques souvent théoriques, j'ai ici intégré une solution de tests complète et développé un dashboard qui a une utilité réelle pour l'équipe technique : monitorer la santé de l'application.

J'ai compris qu'un bon outil ne doit pas seulement fonctionner, il doit être lisible et aider à la prise de décision.

Ce stage m'a appris qu'en tant que développeur, notre rôle n'est pas seulement d'écrire du code, mais de fournir des solutions qui font gagner du temps aux autres.

Signatures