Table des matières

Introduction générale **1**

1 Cadre général du projet **2**
1 Introduction . . . 2
2 Présentation de l'entreprise d'accueil . . . 2
3 Contexte et problématique . . . 3
4 Étude de l'existant . . . 4
4.1 Critères d'évaluation . . . 4
4.2 Solutions étudiées . . . 5
4.3 Synthèse comparative . . . 7
4.4 Analyse critique et justification de la solution . . . 7
5 Solution proposée . . . 8
6 Méthodologie de développement . . . 8
6.1 Choix de la méthodologie . . . 8
6.2 Présentation du processus 2TUP . . . 9
6.3 Application au projet . . . 10
7 Conclusion . . . 10

2 Analyse et spécification des besoins **11**
1 Introduction . . . 11
2 Analyse et spécification des besoins fonctionnels . . . 11
2.1 Identification des acteurs . . . 11
2.2 Expression des besoins fonctionnels . . . 11
2.3 Diagramme général de cas d'utilisation . . . 12
2.4 Cas d'utilisation : Créer un Job de surveillance . . . 14
2.5 Cas d'utilisation : Contrôler le cycle de vie d'un Job . . . 18
2.6 Cas d'utilisation : Consulter l'historique d'exécutions d'un Job . . . 22
3 Analyse et spécification des besoins techniques . . . 25
3.1 Choix techniques . . . 25
4 Analyse et spécification des besoins non fonctionnels . . . 27
5 Conclusion . . . 28

3 Étude conceptuelle **29**
1 Introduction . . . 29
2 Architecture globale du système . . . 29
3 Conception de la base de données . . . 31
3.1 Dictionnaire des entités . . . 31
3.2 Modèle entité-association . . . 34
3.3 Schéma relationnel de la base de données : . . . 35
4 Conception logicielle . . . 35
4.1 Diagramme de classes . . . 35
4.2 Diagrammes de séquences . . . 41
5 Conclusion . . . 43

4 Implémentation **45**
1 Introduction . . . 45
2 Environnement de développement . . . 45
2.1 Environnement matériel . . . 45
2.2 Environnement logiciel . . . 45
3 Architecture technique de l'application . . . 46
4 Déploiement . . . 48
4.1 Diagramme de déploiement . . . 48
4.2 Justification des plateformes de déploiement choisis . . . 48
4.3 Intégration et déploiement continus . . . 49
4.4 Quelques décisions notables lors du déploiement . . . 49
5 Présentation des interfaces . . . 51
5.1 Authentification . . . 51
5.2 Tableau de bord . . . 52
5.3 Création d'un job de surveillance . . . 53
5.4 Gestion des jobs . . . 55
5.5 Suivi des exécutions . . . 56
5.6 Gestion des datapoints . . . 57
5.7 Détection des changements . . . 59
6 Tests et validation . . . 60
6.1 Tests unitaires . . . 60
6.2 Tests d'intégration . . . 61
6.3 Bilan . . . 62
7 Conclusion . . . 62

Conclusion et perspectives **63**

Table des figures

1.1 Logo de VISIOAD 2
1.2 Taux de modification sur 30 jours des pages de tarification SaaS les plus surveillées 3
1.3 Interface de monitoring de Visualping 5
1.4 Page d'accueil de Distill.io 6
1.5 Page d'accueil de Browse AI 6
1.6 Interface de changedetection.io 7
1.7 Cycle de développement en Y du processus 2TUP 9

2.1 Diagramme général des cas d'utilisation de la plateforme 13
2.2 Diagramme de CU « Créer un job de surveillance » 14
2.3 Diagramme de séquence « Créer un job de surveillance » 15
2.4 Maquette IHM : Formulaire de création d'un job de surveillance 17
2.5 Maquette IHM : Outil de sélection manuelle des points de données (Mini-navigateur) 18
2.6 Diagramme du CU « Contrôler le cycle de vie d'un job » 19
2.7 Diagramme de séquence « Contrôler le cycle de vie d'un job » 20
2.8 Maquette IHM : Page de détail d'un job et boutons de contrôle 22
2.9 Maquette IHM : Boîte de dialogue de confirmation de suppression 22
2.10 Diagramme du CU « Consulter l'historique d'exécutions d'un job » 23
2.11 Maquette IHM : Page de l'historique des exécutions d'un job 25

3.1 Architecture globale du système 30
3.2 Modèle entité-association de la base de données 34
3.3 Diagramme de classes des entités du domaine 36
3.4 Diagramme de classes de la couche métier 37
3.5 Diagramme de classes du service de planification 40
3.6 Diagramme de classes du service d'extraction 41
3.7 Diagramme de séquence: Créer un job de surveillance 42
3.8 Diagramme de séquence: Exécution d'un job 43

4.1 Architecture technique de l'application 47

4.2 Diagramme de déploiement de l'application 48
4.3 Problème de domaine : le cookie `accessToken` 50
4.4 Solution par proxy: le cookie est réémis sous `vercel.app` 51
4.5 Interfaces d'authentification 52
4.6 Tableau de bord — liste des cibles surveillées 52
4.7 Interface de création d'un job de surveillance 54
4.8 Outil de sélection visuelle des éléments à extraire 55
4.9 Page de détail d'un job 56
4.10 Historique des exécutions d'un job 56
4.11 Détail d'une exécution — résultats et logs 57
4.12 Liste des datapoints 58
4.13 Détail d'un datapoint avec aperçu en temps réel 58
4.14 Vue de comparaison — changement détecté entre deux exécutions 59

Liste des tableaux

1.1 Coordonnées de VISIOAD 2
1.2 Comparaison des solutions existantes 7

2.1 Description du CU « Créer un job de surveillance » 16
2.2 Description textuelle du CU « Contrôler le cycle de vie d'un job » 21
2.3 Description du CU « Consulter l'historique d'exécutions d'un job » 24

3.1 Tableau descriptif des entités métier de la plateforme 32
3.2 Rôle des classes de la couche métier 38

4.1 Environnement logiciel du projet 46
4.2 Suites de tests unitaires 60
4.3 Suites de tests d'intégration 61
4.4 Couverture de tests par module 62

Remerciements

Je tiens à exprimer ma gratitude à l'équipe de **VISIOAD** pour son soutien constant et leurs conseils pendant mon stage. Un merci particulier à **M. Ghassen Abbes**, mon superviseur, pour sa patience et ses encouragements.

Je souhaite également remercier mon encadrant, **Dr Sami Bhiri**, pour son suivi, ses orientations et ses conseils avisés tout au long de ce projet de fin d'études.

Merci à tous pour cette expérience enrichissante.

Introduction générale

Le web est un espace décentralisé, vaste, et dynamique. Le contenu et les données changent sans cesse, à un rythme toujours croissant. Rien ne reste identique depuis votre dernière visite. Les pages sont mises à jour, les prix fluctuent, les stocks montent et descendent, la publicité est omniprésente.... Face à ces changements constants, un besoin réel se fait jour: **comment sélectionner les changements dont on souhaite être informé, sans avoir à vérifier manuellement à chaque fois ?**

Les entreprises seraient également ravies de disposer d'une solution. Être informé des changements sur les sites web de leurs concurrents leur confère un avantage stratégique considérable. Choisir les données spécifiques à suivre facilitera leur prise de décision et la rendra plus rapide et stratégique. Quant aux particuliers, ils souhaiteraient être informés des nouveautés concernant leurs produits, contenus et blogs préférés, sans avoir à vérifier manuellement et sans s'abonner à des newsletters bourrées d'informations inutiles.

Les solutions existantes restent soit trop génériques, soit trop techniques pour un usage courant. L'utilisateur se retrouve alors contraint de surveiller manuellement les pages qui l'intéressent, ou de déléguer cette tâche à des outils rigides qu'il ne maîtrise pas.

C'est dans ce contexte que s'inscrit ce projet. Sa mission est de développer une plateforme qui permet à l'utilisateur de définir les URLs à surveiller, de configurer la fréquence et la méthode d'extraction des données, et de choisir quand et comment il veut être notifié. L'utilisateur conserve ainsi le contrôle sur ce qu'il surveille, quand il le surveille, et comment il en est informé.

Chapitre 1

Cadre général du projet

1 Introduction

Le premier chapitre, qui présente le cadre générale du projet, commence par une introduction à l'entreprise d'accueil, suivie de la motivation à l'origine du projet (le problème principal et son utilité potentielle). Ensuite, les solutions existantes, que nous analyserons afin de mieux comprendre le marché. Enfin, nous discuterons la méthodologie adoptée pour le développement du projet.

2 Présentation de l'entreprise d'accueil

VISIOAD est une agence de marketing digital fondée en 2022, basée à Sousse, Tunisie.

Figure 1.1. *Logo de VISIOAD*

Table 1.1. *Coordonnées de VISIOAD*

Adresse	Immeuble Centre Ibrahim, Av. Habib Bourguiba, Sousse 4000
Ville	Sousse, Tunisie
Téléphone	+216 31 439 350
Site web	https://www.visioad.com/

L'entreprise propose des services de marketing digital : gestion de campagnes publicitaires, référencement SEO/SEA, developpement web et mobile, et création de contenu multimédia. Elle intervient sur le conseil strategique, l'implémentation technique et la gestion de campagnes, pour des clients locals et internationals. VISIOAD se distingue par une expertise dans le secteur agricole américain, là où elle accompagne des acteurs de l'Agri-Tech dans leur transformation digitale.

3 Contexte et problématique

Dans un échantillon de plus de 9 000 pages web surveillées par Visualping (une solution présentée dans la section suivante), 42 % ont signalé au moins une modification en moins de 30 jours. De plus, « les pages de plans et tarifs présentent la plus forte densité de signal parmi tous les types de contenu surveillés ». Voici quelques exemples de pages parmi les plus consultées de cet échantillon :

SaaS pricing page	Active monitors in sample	30-day change rate
Zoom	18	100%
HubSpot	88	96%
Constant Contact	23	96%
Datadog	61	89%
DocuSign	29	83%
Stripe	20	80%
Twilio	44	77%
Salesforce	59	76%
Klaviyo	30	73%
Monday.com	29	69%

Figure 1.2. *Taux de modification sur 30 jours des pages de tarification SaaS les plus surveillées*

Source : Visualping, échantillon de 9 705 surveillances actives [1].

En résumé, que vous soyez un particulier soucieux des prix et souhaitiez être informé de leur évolution, ou une entreprise désireuse de suivre les pages de tarification de ses concurrents, vous devez disposer d'un outil plus performant qu'une simple surveillance manuelle. Cette solution répond à votre besoin quotidien de façon automatisé et fluide.

Cependant, en matière de surveillance de sites web, on se retrouve généralement face à deux options :

- La surveillance manuelle, qui, comme mentionnée précédemment, est impossible à maintenir sur le long terme

▷ Les scripts d'extraction, exécutés à la demande, ciblés sur des sites web spécifiques et isolés, qui ne permettent ni analyse ni historique des changements

La surveillance automatique du web présente également des contraintes inhérent, tels que la détection de bots, la difficulté de filtrer le bruit (le web étant complexe et encombré de publicités), et enfin, la spécificité de chaque site web; ils existent plusieurs architectures web, et certaines données ne sont accessibles que via des manipulations particulières, souvent difficiles à simuler par des scripts.

Ces limitations nous permettent déjà de cerner les principaux axes de notre projet. Pour être fonctionnel et utile, il doit :

▷ Fonctionner sur différentes architectures web

▷ S'adapter aux restrictions

▷ Simuler certaines actions du navigateur

▷ Filtrer les bruit lors de l'extraction

L'objectif se précise : créer une plateforme d'extraction et de suivi automatisé des données du contenu web, capable de surmonter les difficultés mentionnées, tout en étant fiable, maintenable et adaptée aux besoins du marché.

4 Étude de l'existant

4.1 Critères d'évaluation

Suite à la problématique abordée dans la section précédente, nous pouvons conclure sur quelques critères d'évaluation des solutions existantes :

▷ **Interface conviviale** : elle doit masquer la complexité des opérations techniques effectuées en arrière-plan.

▷ **Visualiseur de différences avancé** : visualiser les différences entre deux snapshots est le cœur fonctionnel du besoin.

▷ **Alertes configurables** : notifier l'utilisateur selon des options définies (ex. : canaux de notification, conditions de déclenchement).

▷ **Support des architectures web modernes** : les sites e-commerce adoptent des architectures et des approches d'implémentation très variées.

▷ **Filtrage du bruit DOM** : exclure les changements non pertinents (publicités..).

▷ **Gestion anti-bot / proxies** : contourner les protections des sites surveillés.

4.2 Solutions étudiées

Les quatre solutions ci-dessous ont été sélectionnées car elles représentent les approches dominantes du marché : deux plateformes SaaS commerciales grand public (Visualping, Distill.io), une plateforme IA no-code (Browse AI), et une solution open-source auto-hébergée (changedetection.io). Ensemble, elles couvrent l'ensemble du spectre des stratégies disponibles.

Visualping Visualping détecte les changements par comparaison de captures d'écran visuelles. Elle envoie des alertes par email, SMS ou Slack. La prise en main est rapide grâce à une interface drag-and-drop. Cependant, la détection est purement visuelle : elle ne produit pas de diff structuré du DOM et n'expose pas les données extraites. Elle ne répond donc pas au besoin d'analyse différentielle granulaire.

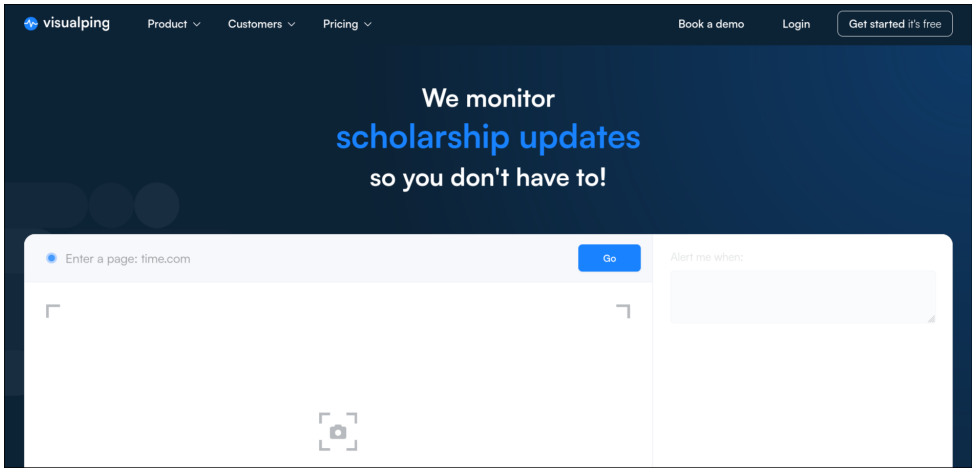


Figure 1.3. *Interface de monitoring de Visualping*

Distill.io Distill.io surveille des pages via des sélecteurs XPath, CSS ou des expressions régulières. La version cloud et l'extension navigateur couvrent les cas courants. La limite principale est l'absence de visualisation comparative des snapshots historiques. La version gratuite est restreinte à 25 pages, et la gestion des proxies n'est pas intégrée.

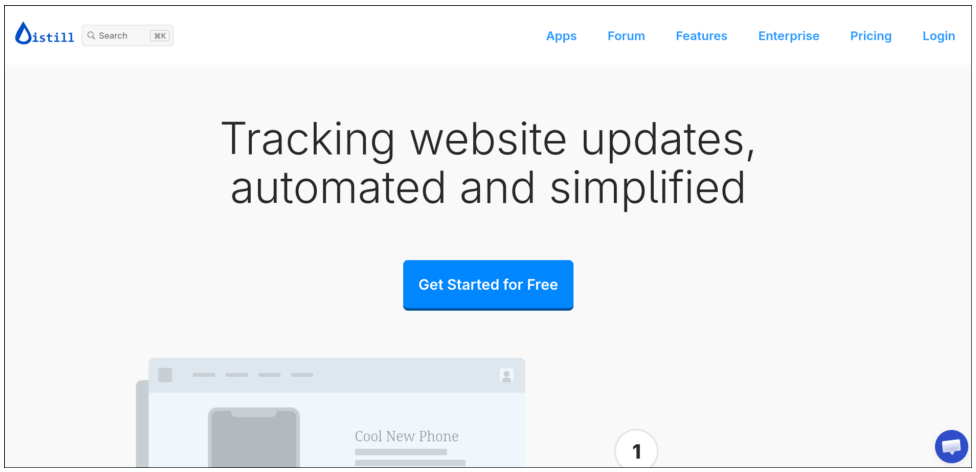


Figure 1.4. *Page d'accueil de Distill.io*

Browse AI Browse AI est une plateforme no-code alimentée par l'IA. Elle s'adapte automatiquement aux changements de structure de sites et intègre un contournement anti-bot. Ses capacités techniques sont solides, mais son coût est prohibitif pour un usage professionnel intensif. Surtout, elle ne permet pas de personnaliser les algorithmes de détection ni de contrôler l'infrastructure sous-jacente.

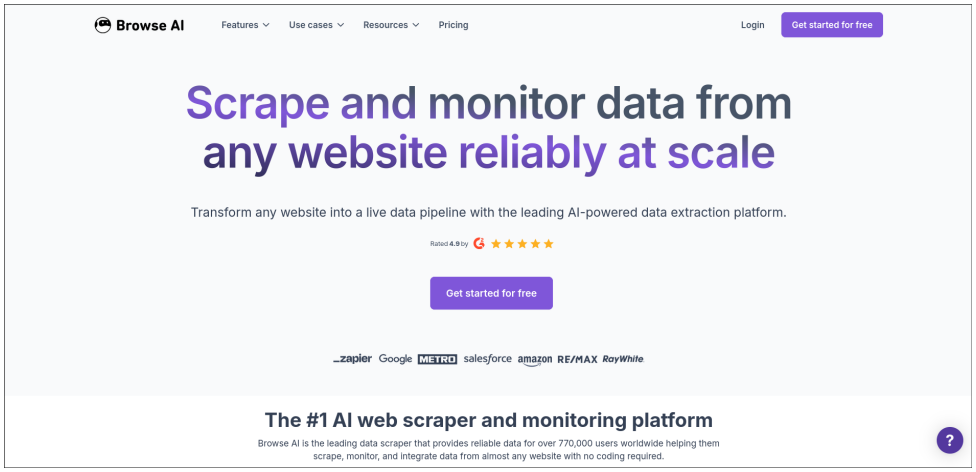


Figure 1.5. *Page d'accueil de Browse AI*

changedetection.io changedetection.io est une solution open-source auto-hébergée. Elle supporte le monitoring JSON/API, les webhooks, et s'intègre avec Playwright pour les sites JavaScript. Elle offre flexibilité et gratuité, mais exige une infrastructure propre, des compétences DevOps pour la mise en production, et son interface ne propose pas de diff viewer professionnel.

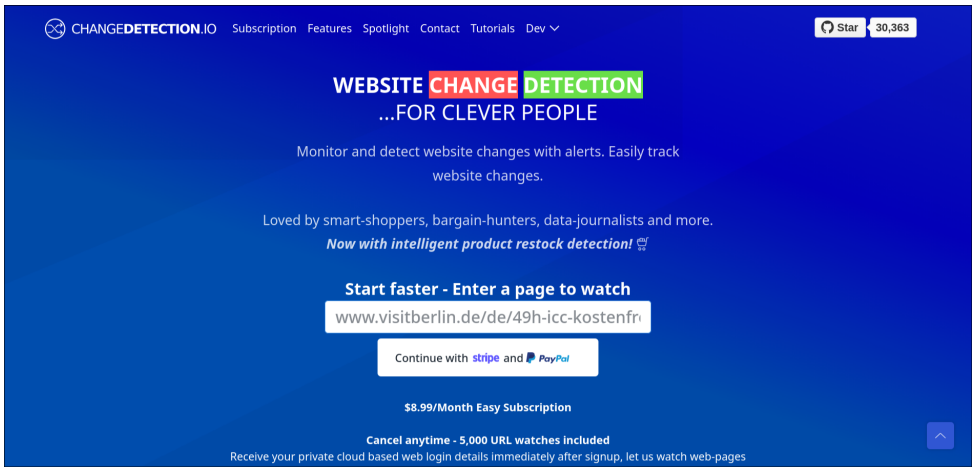


Figure 1.6. *Interface de changedetection.io*

4.3 Synthèse comparative

Le tableau 1.2 présente la comparaison des solutions selon les critères définis.

Table 1.2. *Comparaison des solutions existantes*

Solution / Critère	Visualping	Distill.io	Browse AI	changedetection.io
Interface conviviale	✓	✓	✓	×
Visualiseur de différences	×	○	×	×
Alertes configurables	✓	✓	✓	✓
Support architectures modernes	✓	✓	✓	✓
Filtrage du bruit DOM	○	○	✓	×
Contournement des protections	✓	×	✓	○
Coût mensuel	10–14 $	15 $	Élevé	Gratuit*

✓ : disponible × : absent ○ : partiellement disponible ou limité
* : Open-source, mais nécessite une infrastructure d'hébergement.

4.4 Analyse critique et justification de la solution

Aucune des solutions étudiées ne couvre simultanément les trois besoins clés :

1. Un support des différentes architectures web.

2. Un filtrage du bruit DOM robuste pour isoler les changements pertinents.

3. Un visualiseur de différence structuré et exploitable.

Les plateformes commerciales sacrifient la granularité au profit de la simplicité. Browse AI offre des capacités techniques avancées, mais à des coûts élevés. changedetection.io est flexible mais nécessite une expertise opérationnelle significative et n'expose pas d'interface professionnelle.

Ce constat justifie le développement d'une solution sur mesure.

5 Solution proposée

Notre solution proposée est une plateforme d'extraction de données et de surveillance de changements sur le web. Elle permet à l'utilisateur de définir les données à suivre sur les URL de son choix et de configurer la fréquence à laquelle la plateforme vérifie automatiquement les modifications (mensuelle, hebdomadaire, quotidienne). L'utilisateur est ensuite informé lorsqu'un changement est détecté.

Ses principales fonctionnalités sont :

- ▷ Création de jobs: il s'agit du principal cas d'utilisation, l'utilisateur définit les données à suivre, les URL cibles ainsi que les paramètres d'extraction et de notification
- ▷ Gestion des paramètres des jobs: l'utilisateur peut consulter l'ensemble de ses jobs, visualiser les données suivies et les URL cibles, puis modifier les paramètres associés si nécessaire
- ▷ Plusieurs stratégies d'extraction: selon les caractéristiques du site web ciblé, l'utilisateur dispose de différentes méthodes d'extraction (extraction basique pour les pages statiques, extraction avancée pour les sites dynamiques et complexes..).
- ▷ Notifications: la plateforme notifie l'utilisateur (si activé) à la fin de chaque exécution et affiche les résultats de l'extraction, les modifications du contenu, et les détails associés.
- ▷ Historique des exécutions : l'utilisateur peut consulter l'historique de chaque exécution, même celles ayant échoué.

Cette plateforme s'adresse aussi bien aux utilisateurs individuels qu'aux entreprises. L'accès aux fonctionnalités nécessite la création d'un compte utilisateur.

6 Méthodologie de développement

6.1 Choix de la méthodologie

L'entreprise d'accueil travaille selon le Processus Unifié et a orienté le choix des technologies dès le début du projet. Les contraintes techniques étaient donc connues avant de détailler les besoins fonctionnels.

Nous avons adopté le processus **2TUP** (*2 Track Unified Process*), qui s'inscrit dans le Processus Unifié et convient à cette situation : il est itératif, mais sépare explicitement l'axe technique de l'axe fonctionnel. Les deux branches se développent en parallèle, puis fusionnent en phase de réalisation.

La nature du projet renforce ce choix. La solution doit interagir avec des sites de natures très différentes : pages statiques, applications JavaScript, sites protégés. Chaque type de site peut nécessiter des frameworks et des stratégies d'extraction distinctes. L'axe technique évolue donc au fil des itérations, indépendamment de la logique fonctionnelle de surveillance.

6.2 Présentation du processus 2TUP

Roques et Vallée définissent le 2TUP comme un processus unifié fondé sur la séparation de toute évolution du système d'information en deux axes traités en parallèle : un axe fonctionnel et un axe technique [2].

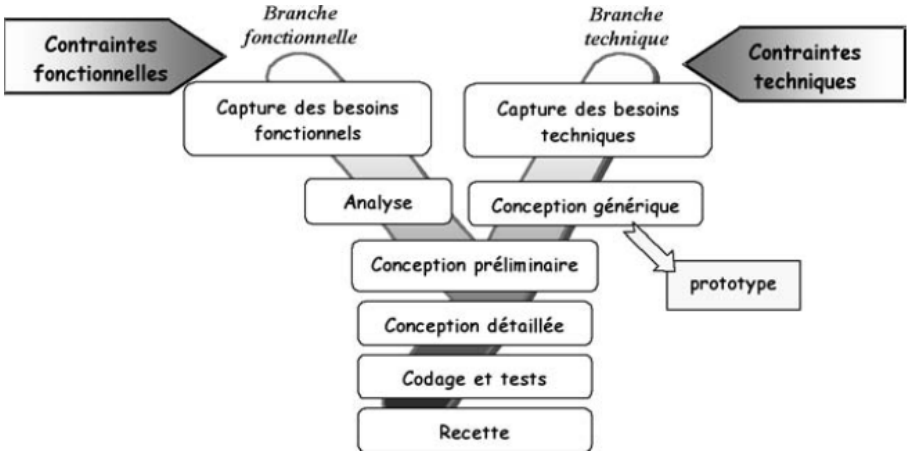


Figure 1.7. *Cycle de développement en Y du processus 2TUP*
Source : *UML 2 en action* [2]

2TUP propose un **cycle en Y** articulé autour de trois phases [2] :

- **Branche fonctionnelle (gauche du Y)** : capture des besoins fonctionnels indépendamment de toute contrainte technique. Elle produit un modèle centré sur le métier des utilisateurs finaux.
- **Branche technique (droite du Y)** : définition de l'architecture et des choix technologiques indépendamment des fonctions à réaliser.
- **Phase de réalisation (bas du Y)** : fusion des deux branches. Elle produit la conception applicative et la livraison de la solution.

6.3 Application au projet

Branche fonctionnelle. Nous définissons et modélisons les besoins via des diagrammes de cas d'utilisation, des descriptions textuelles et des maquettes d'interface.

Branche technique. Nous définissons l'architecture technique du projet et les choix de frameworks et bibliothèques.

Phase de réalisation. Nous fusionnons les deux branches pour produire la conception logicielle, l'implémentation et les tests et validations.

7 Conclusion

Ce chapitre a présenté le cadre général du projet : l'entreprise d'accueil, la problématique à l'origine de la solution, une analyse des outils existants, et la solution retenue. Le point fort de notre projet réside dans sa capacité à gérer de multiples architectures de sites web. Et comme le marché utilise principalement ce type de solutions pour le suivi des prix et l'analyse concurrentielle, notre solution doit être particulièrement adaptée à ces usages.

Chapitre 2

Analyse et spécification des besoins

1 Introduction

En suivant le cadre de processus unifié de développement, nous abordons par la première étape du développement d'un projet, qui est « l'analyse et la spécification des besoins ».

Dans ce chapitre, nous identifions l'acteur impliqué et exprimons les besoins fonctionnels, que nous modélisons à travers un diagramme général de cas d'utilisation. Trois cas d'utilisation majeurs sont ensuite détaillés : chacun est accompagné d'un diagramme de cas d'utilisation, d'un diagramme de séquence, d'une description textuelle et de maquettes d'interface. Nous terminons par la présentation des choix technologiques et des exigences non fonctionnelles.

2 Analyse et spécification des besoins fonctionnels

2.1 Identification des acteurs

La plateforme ne compte qu'un seul acteur interagissant avec le système:

- **Utilisateur** : L'acteur principal. Une fois authentifié, il dispose d'un espace lui permettant de gérer les jobs, les URL cibles, les points de données, les exécutions et les paramètres de profil.

2.2 Expression des besoins fonctionnels

Nous détaillons les cas d'utilisations de l'utilisateur comme suit:

▷ **Créer un Job de surveillance**: L'utilisateur crée une job en définissant l'URL cible, les points de données (élément(s) HTML), la stratégie et la fréquence d'extraction, le format de sortie et les paramètres de notification.

▷ **Gérer les Jobs**: L'utilisateur consulte, modifie ou supprime les jobs.

- ▷ **Gérer les points de données**: L'utilisateur consulte, modifie ou supprime les points de données définis sur ses URL-cibles.
- ▷ **Consulter et supprimer les URL-cibles**: L'utilisateur accède à la liste de ses URL-cibles avec leurs métadonnées (nombre de points associés, état des jobs) et peut supprimer les entrées non souhaitées.
- ▷ **Contrôler le cycle de vie d'un job**: L'utilisateur peut suspendre un job actif, ce qui interrompt sa planification, la reprendre, ce qui réactive sa planification, ou l'exécuter immédiatement indépendamment de l'exécution programmée.
- ▷ **Recevoir des notifications**: L'utilisateur est informé à la fin d'une exécution, la détection d'une mise à jour de contenu, ou une erreur lors de l'exécution d'une tâche. Il peut configurer ses préférences afin de recevoir ces notifications par e-mail et/ou via l'application.
- ▷ **Afficher l'historique des exécutions d'un job**: L'utilisateur consulte l'historique des exécutions d'un job, triées par date de création. Il peut également sélectionner et consulter les détails de chaque exécution.
- ▷ **Consulter une exécution** (Résultats et logs): L'utilisateur consulte les détails d'une exécution, les résultats générés, dans le format qui l'a choisi. Ansi que les logs qui sont mis à jour en temps réel.
- ▷ **Gérer son profil**: L'utilisateur met à jour ses informations personnelles et configure ses préférences, telles que les paramètres de notifications ou la modification du mot de passe.

2.3 Diagramme général de cas d'utilisation

Le diagramme de cas d'utilisation suivant identifie les interactions entre les acteurs du système et les fonctionnalités qu'il expose (Figure 2.1). Ce diagramme sert de référence pour que chaque besoin fonctionnel est bien couvert avant de passer à la conception.

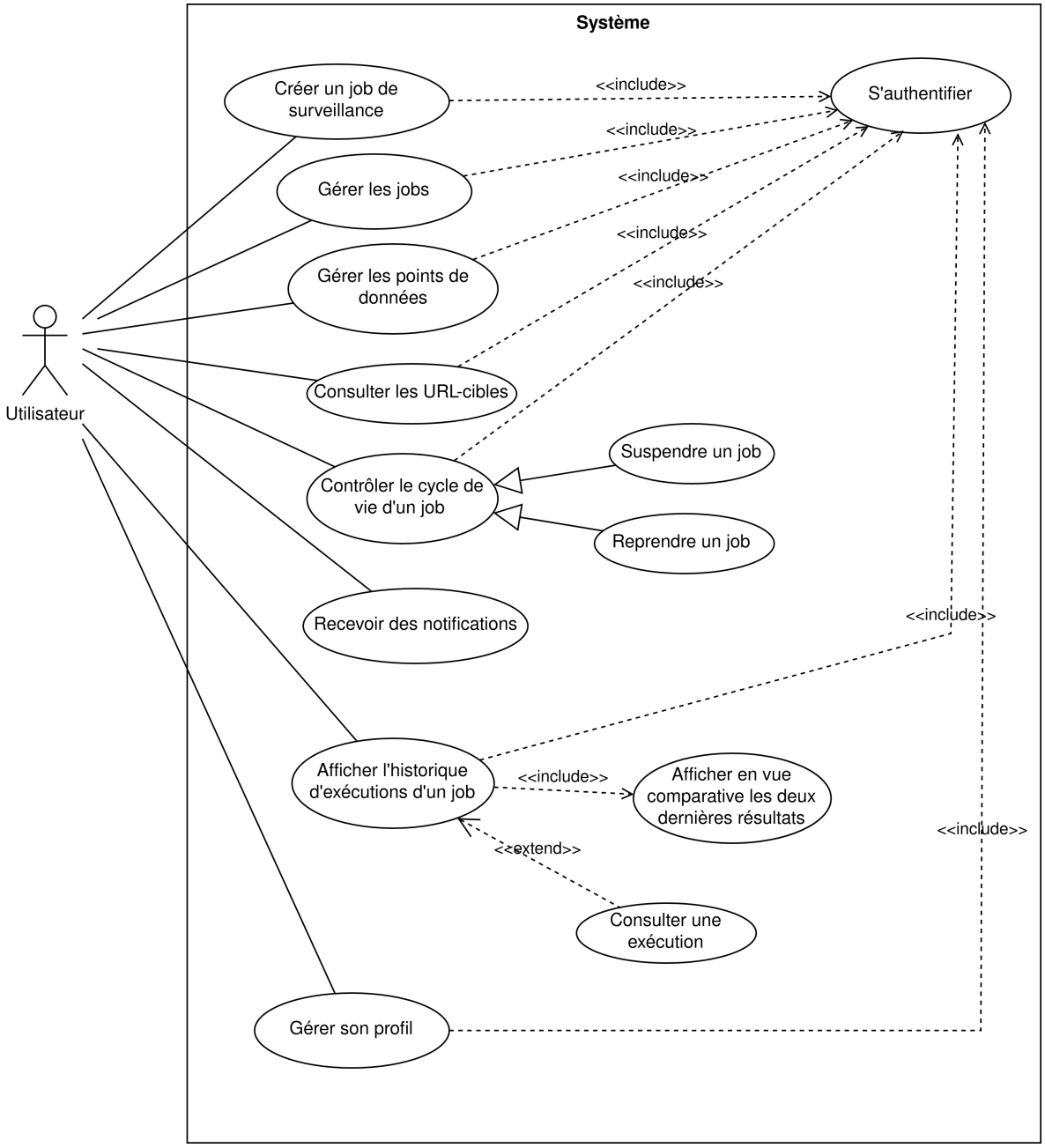


Figure 2.1. *Diagramme général des cas d'utilisation de la plateforme*

Dans la suite de cette section, nous détaillerons les trois cas d'utilisation suivants :

- ▷ Créer un Job de surveillance ;
- ▷ Contrôler le cycle de vie d'un Job ;
- ▷ Afficher l'historique d'exécutions d'un job.

Pour chacun d'entre eux, nous présenterons le diagramme de cas d'utilisation, la description textuelle, le diagramme de séquence du scénario associé, et la maquette de l'interface (IHM) correspondante.

2.4 Cas d'utilisation : Créer un Job de surveillance

La création d'un job est le principal cas d'utilisation de la plateforme. Comme le montre la figure 2.2, ce cas inclut deux autres sous-cas: la définition du point de données et la planification de l'exécution.

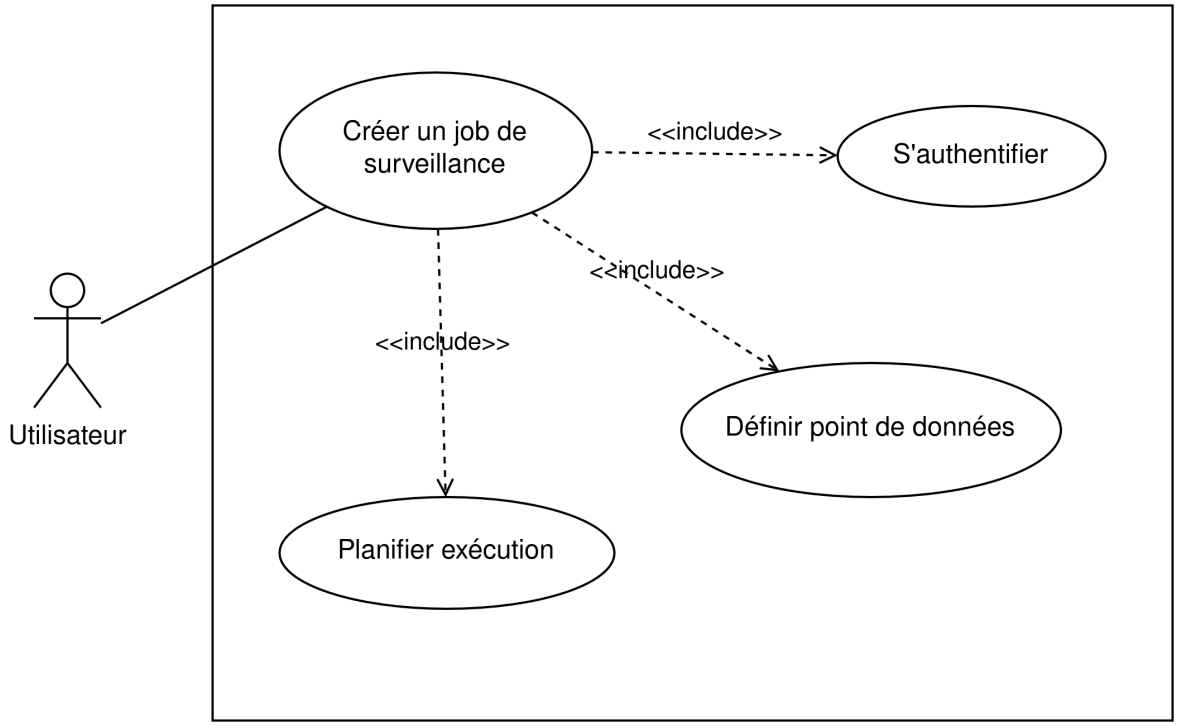


Figure 2.2. *Diagramme de CU « Créer un job de surveillance »*

Diagramme de séquence : Création d'un job

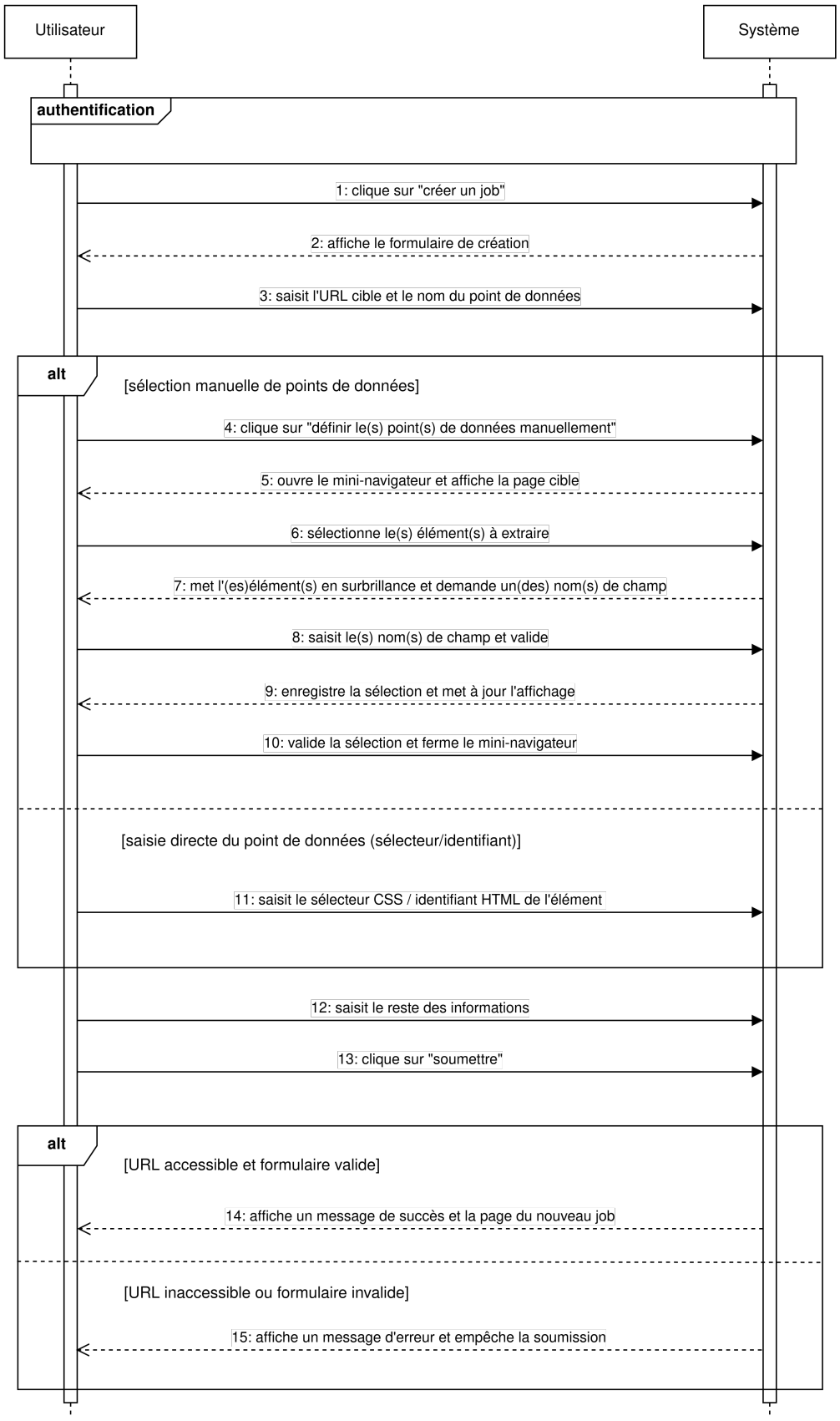


Figure 2.3. *Diagramme de séquence « Créer un job de surveillance »*

Description textuelle

Table 2.1. *Description du CU « Créer un job de surveillance »*

Cas d'utilisation	Créer un job de surveillance
Acteur	Utilisateur
Précondition	L'utilisateur est authentifié sur la plateforme.
Description du scénario principal	1. L'utilisateur clique sur « Créer un job ». 2. Le système renvoie le formulaire. 3. Il saisit l'URL de la page à surveiller. 4. Il définit le point de données à suivre, soit manuellement via un mini-navigateur intégré, soit en saisissant directement son identifiant. 5. Il choisit le type de planification : exécution unique ou récurrente (horaire, journalière, hebdomadaire). 6. Il sélectionne le type d'extracteur et le format de sortie souhaité. 7. Il configure ses préférences de notification : fin d'exécution, différence détectée, échec. 8. Il soumet le formulaire. 9. Le système crée l'URL-cible (ou réutilise une existante), le point de données et le job associé, puis le planifie.
Post-condition	Le job est créé et planifié ; il sera exécuté automatiquement selon la configuration définie.
Exception	Si l'URL saisie est inaccessible, le système affiche un message d'erreur et empêche la soumission du formulaire.

Maquettes de l'Interface Utilisateur (IHM)

Afin de mieux comprendre le cas d'utilisation de la création de Jobs et de mieux communiquer avec le client, il semble judicieux de créer une maquette. Même si celle-ci n'a pas besoin d'être définitive, elle permettra d'expliquer les étapes générales nécessaires à la création d'un Job.

La Figure 2.4 illustre le formulaire centralisé regroupant les configurations et étapes de création du job. La Figure 2.5 présente l'idée du mini-navigateur intégré permettant la sélection manuelle des éléments HTML à surveiller.

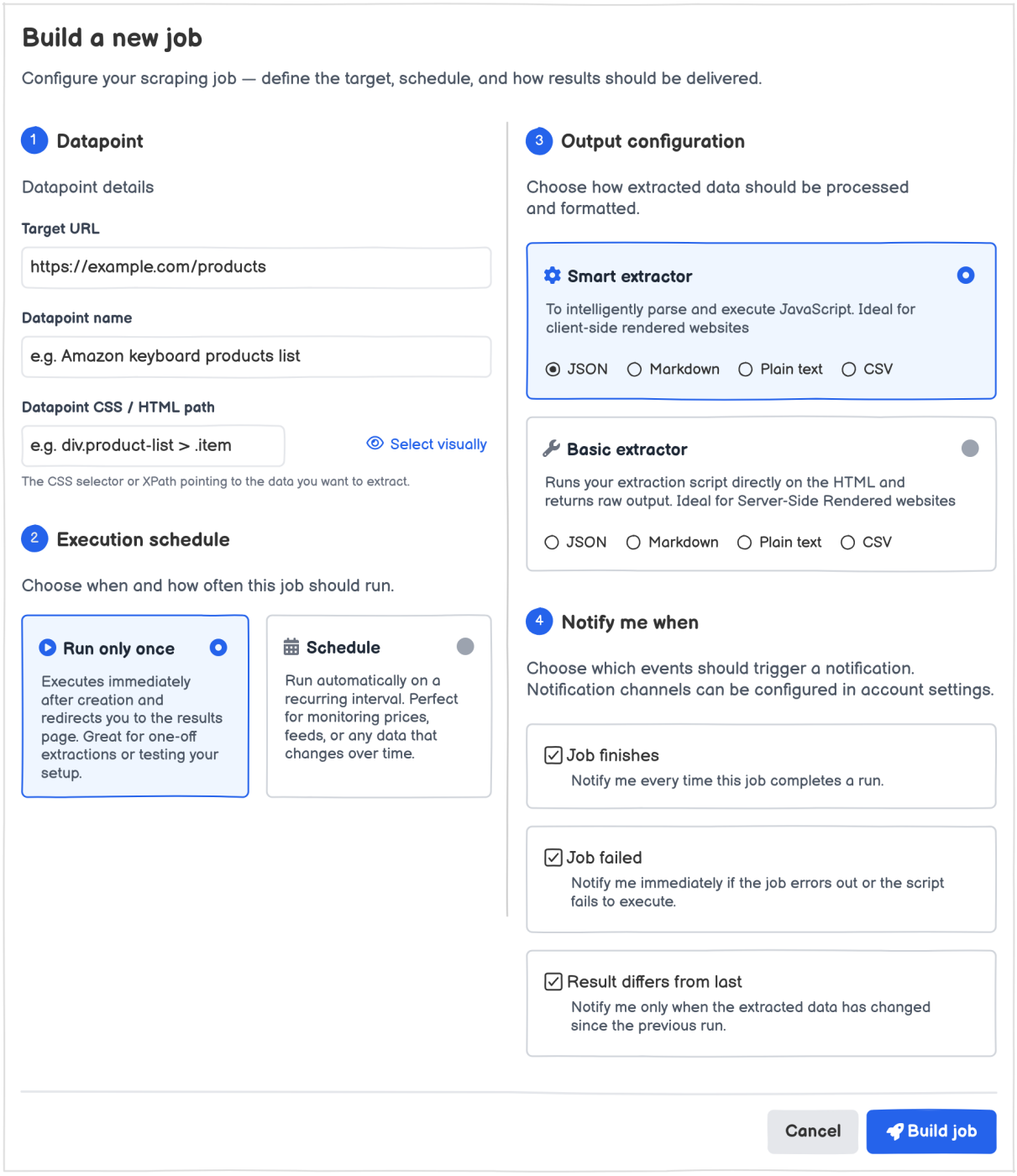


Figure 2.4. *Maquette IHM : Formulaire de création d'un job de surveillance*

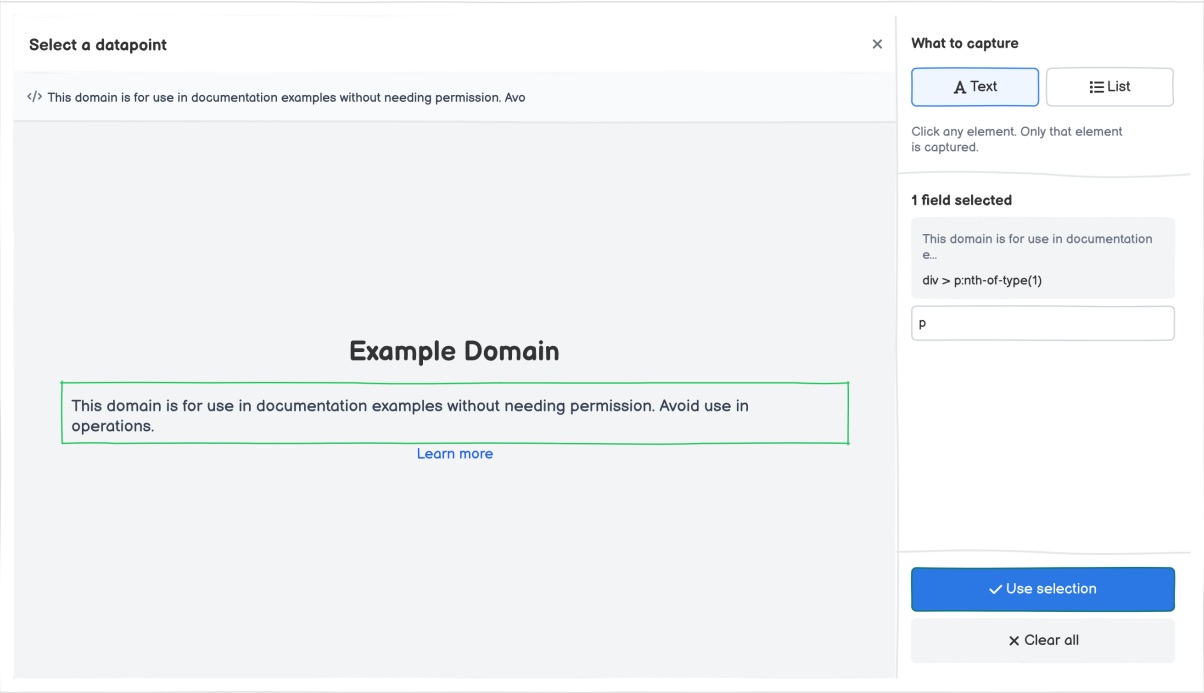


Figure 2.5. *Maquette IHM : Outil de sélection manuelle des points de données (Mini-navigateur)*

2.5 Cas d'utilisation : Contrôler le cycle de vie d'un Job

Une fois un job créé, l'utilisateur doit pouvoir agir sur son état d'exécution. La figure 2.6 présente le diagramme de cas d'utilisation « Contrôler le cycle de vie d'un job ». Ce cas d'utilisation permet à l'utilisateur de gérer les jobs précédemment créés. Il regroupe plusieurs actions fondamentales : la mise en pause d'une surveillance, sa reprise, son exécution manuelle immédiate, ainsi que sa suppression.

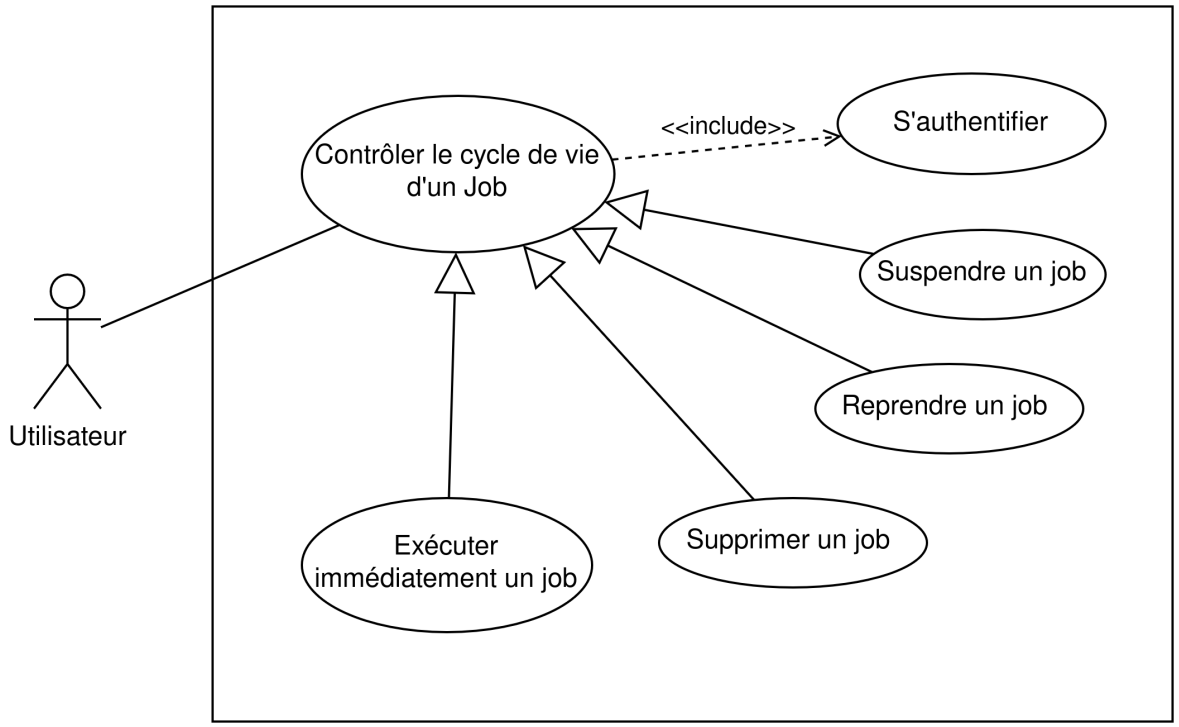


Figure 2.6. *Diagramme du CU « Contrôler le cycle de vie d'un job »*

Diagramme de séquence : Contrôler le cycle de vie d'un Job

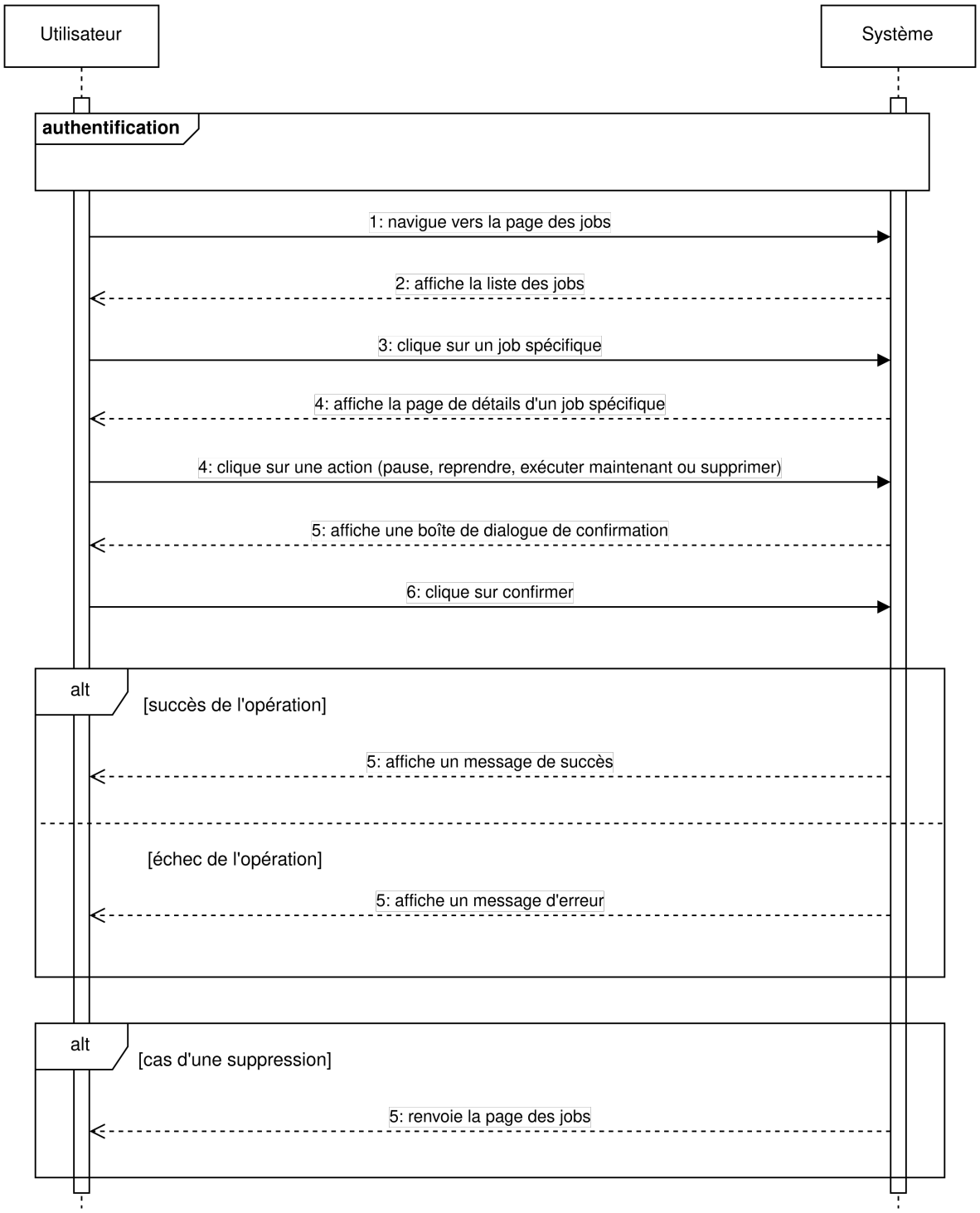


Figure 2.7. *Diagramme de séquence « Contrôler le cycle de vie d'un job »*

Description textuelle

Table 2.2. *Description textuelle du CU « Contrôler le cycle de vie d'un job »*

Cas d'utilisation	Contrôler le cycle de vie d'un job
Acteur	Utilisateur
Précondition	L'utilisateur est authentifié et possède au moins un job configuré.
Description du scénario principal	1. L'utilisateur accède à la page de gestion des jobs. 2. Le système affiche la liste des jobs avec leurs statuts actuels (Actif, En pause), leurs dates de création, et leur nombre d'exécutions. 3. L'utilisateur choisit un job et accède à la page de détail d'un job spécifique. 4. L'utilisateur sélectionne une action parmi : Mettre en pause, Reprendre, Exécuter maintenant, ou Supprimer. 5. Le système affiche une boîte de dialogue demandant une confirmation explicite. 6. L'utilisateur confirme son action. 7. Dans le cas d'une suppression, le système renvoie l'utilisateur à la page de gestion des jobs. 8. Le système met à jour l'état du job et affiche un message de succès.
Post-condition	L'état du job est mis à jour selon l'action choisie (suspendu, réactivé, exécuté ou supprimé).

Maquettes de l'Interface Utilisateur (IHM)

Plutôt que quatre pages séparées pour chaque cas d'utilisation, une seule page de détail expose les actions disponibles selon l'état courant du job, accompagnées d'une confirmation explicite pour les opérations irréversibles La figure 2.8 présente la maquette de l'interface détaillée d'un job. Celle-ci met en évidence son statut en temps réel, ses attributs, ainsi que les boutons d'actions permettant de contrôler son cycle de vie (pause, exécution manuelle, suppression). La figure 2.9 présente la boîte de dialogue de confirmation, un mécanisme ergonomique pour éviter la suppression accidentelle d'un job.

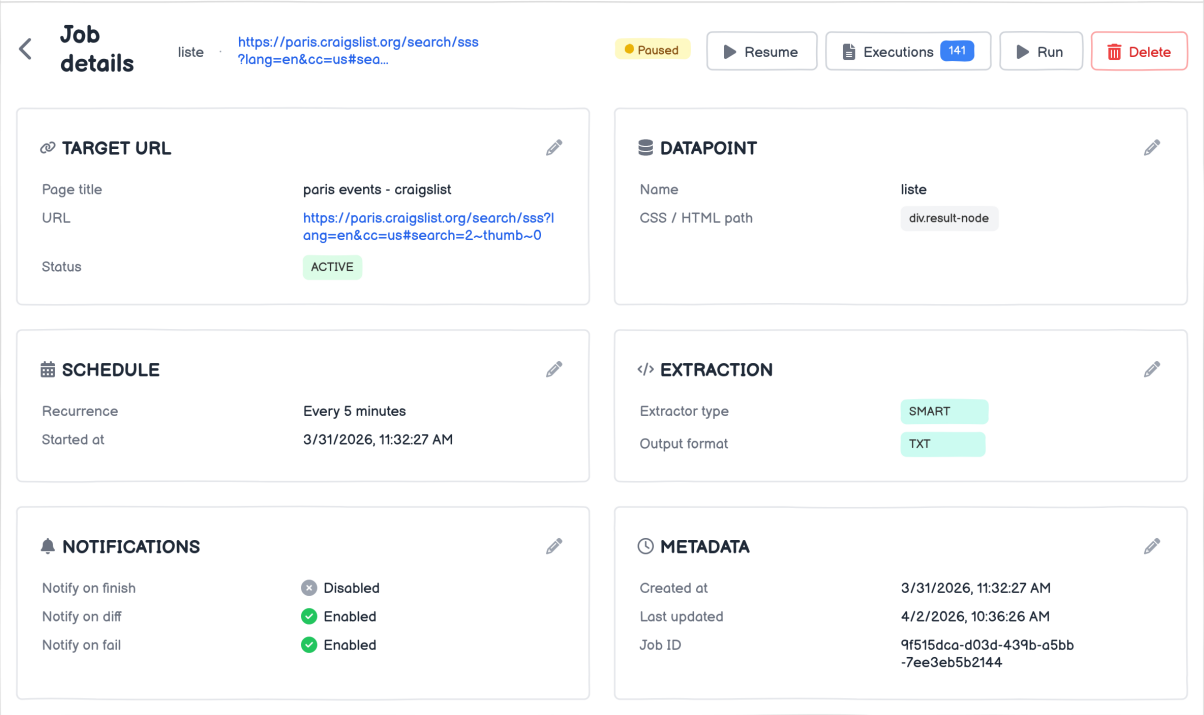


Figure 2.8. *Maquette IHM : Page de détail d'un job et boutons de contrôle*

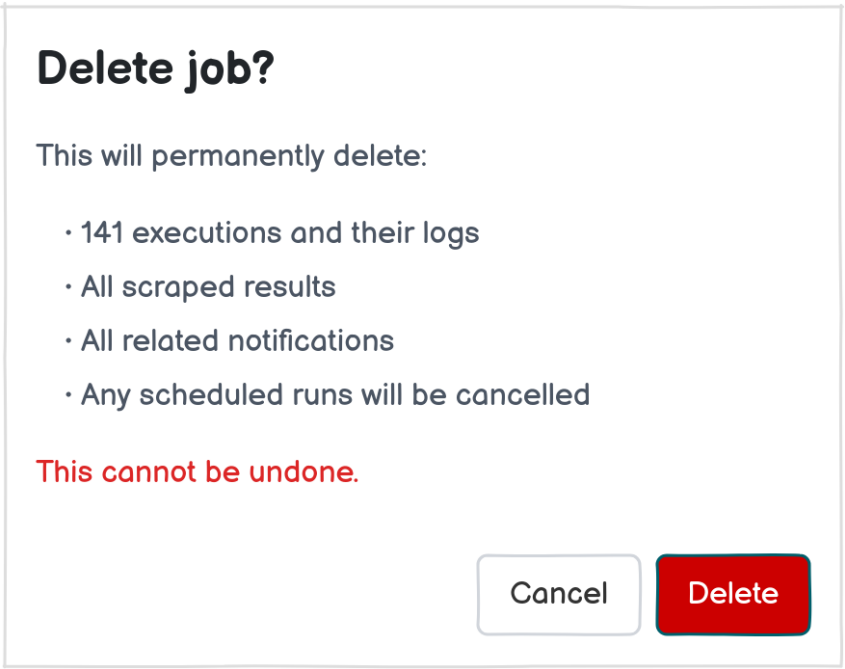


Figure 2.9. *Maquette IHM : Boîte de dialogue de confirmation de suppression*

2.6 Cas d'utilisation : Consulter l'historique d'exécutions d'un Job

Ce cas d'utilisation permet à l'utilisateur de consulter l'historique complet des exécutions. Si au moins deux exécutions sont disponibles, le système propose une

comparaison automatique des résultats des deux dernières, côte à côte, mettant en évidence les différences du contenu extrait. L'utilisateur peut également consulter chaque exécution individuellement pour accéder aux logs et aux résultats générés. La figure 2.10 illustre ce fonctionnement.

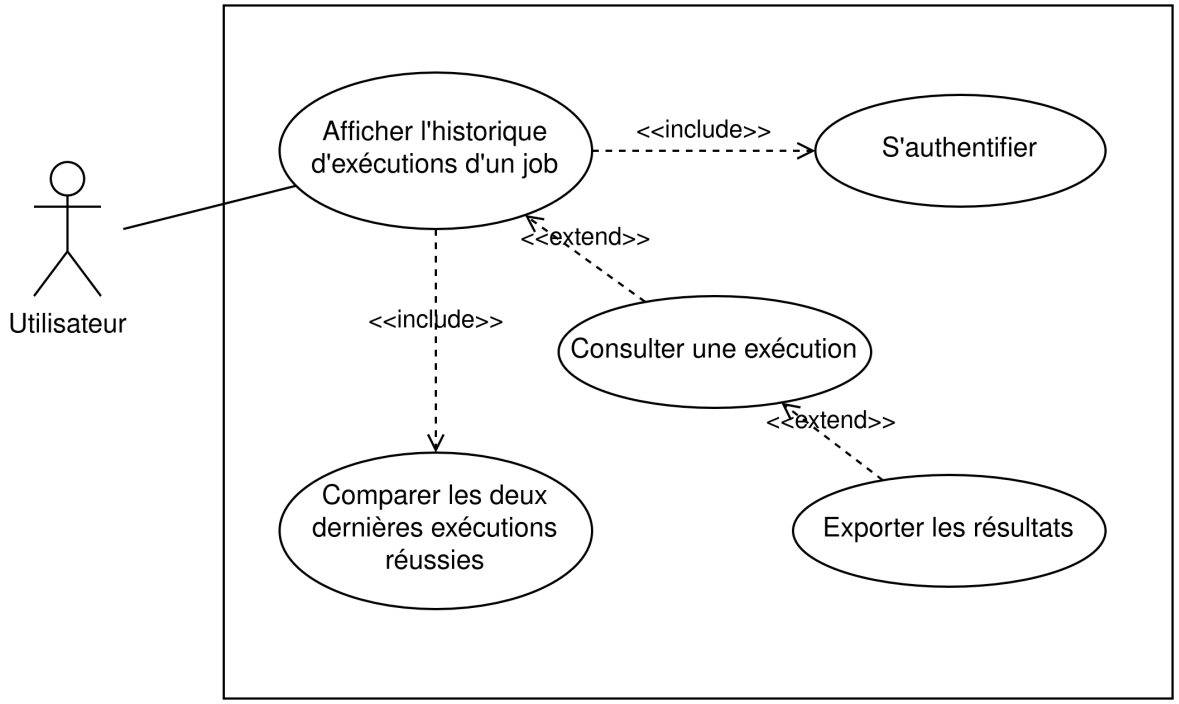


Figure 2.10. *Diagramme du CU « Consulter l'historique d'exécutions d'un job »*

Description textuelle

Table 2.3. *Description du CU « Consulter l'historique d'exécutions d'un job »*

Cas d'utilisation	Consulter l'historique d'exécutions d'un job
Acteur	Utilisateur
Précondition	• L'utilisateur est authentifié. • L'utilisateur dispose d'au moins un job de surveillance configuré.
Description du scénario principal	1. L'utilisateur navigue vers la liste de ses jobs. 2. Il sélectionne un job spécifique pour accéder à ses détails. 3. Il accède à la section dédiée à l'historique. 4. Le système affiche l'historique sous forme de tableau, présentant les métadonnées clés de chaque exécution (statut, date et heure de lancement, durée d'exécution), ainsi qu'une vue comparative exposant les résultats des deux dernières exécutions.
Scénario alternatif	**Historique limité à une seule exécution :** À l'étape 4, si le job ne possède qu'une seule exécution enregistrée, le système n'affiche pas de vue comparative. Il présente directement le tableau d'historique accompagné des détails de cette unique exécution (résultats extraits et logs techniques).
Post-condition	L'utilisateur a pris connaissance de la chronologie des exécutions et, le cas échéant, a pu analyser ou exporter les données d'une exécution précise.

Maquettes de l'Interface Utilisateur (IHM)

La Figure 2.11 présente la maquette de la page d'historique. Elle est structurée en deux zones : en haut, la vue comparative des deux dernières exécutions avec mise en évidence des différences (lignes ajoutées, modifiées ou supprimées) ; en bas, le tableau d'historique listant l'ensemble des exécutions avec leur statut, horodatage, durée et un aperçu du log.

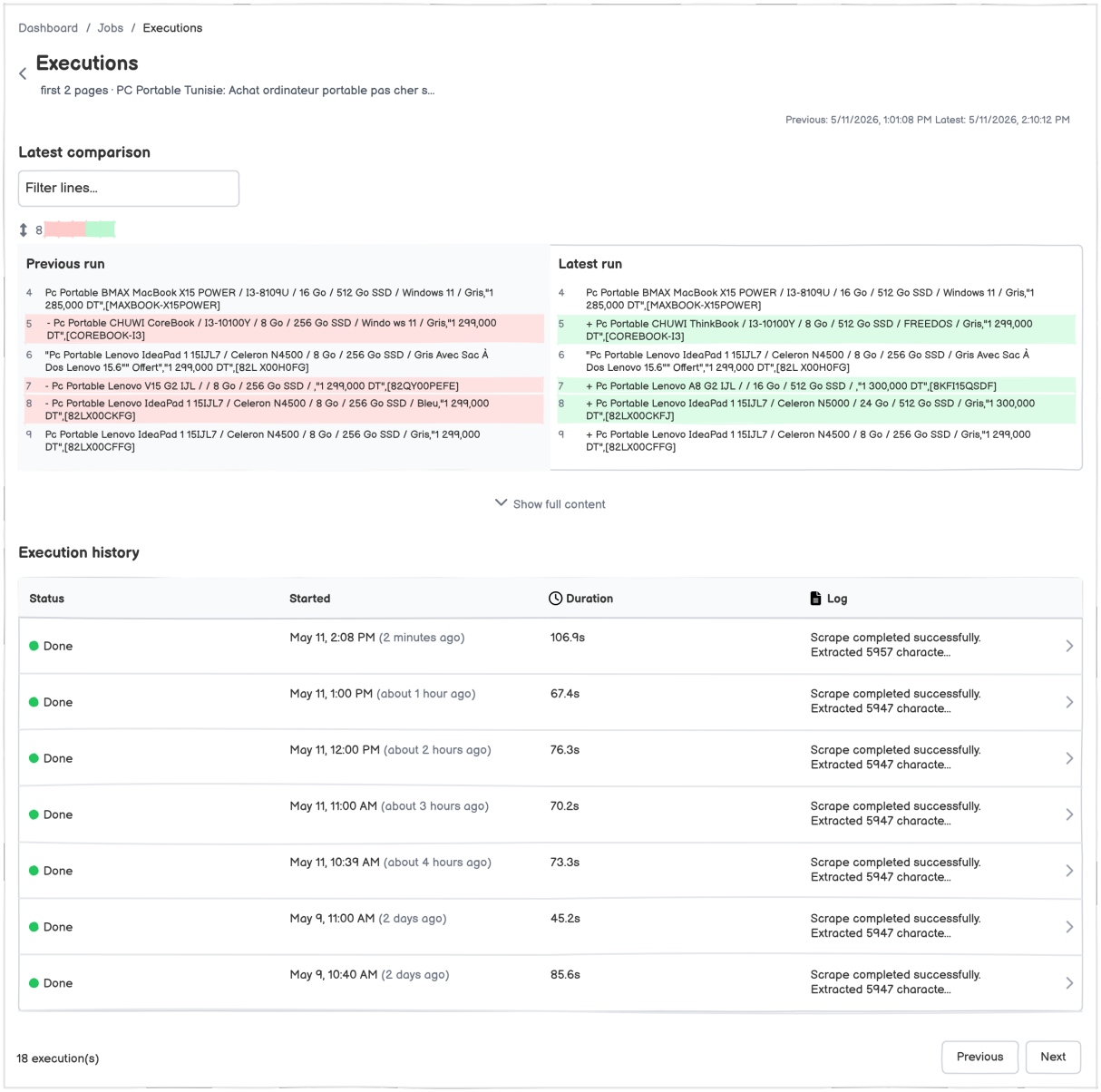


Figure 2.11. *Maquette IHM : Page de l'historique des exécutions d'un job*

3 Analyse et spécification des besoins techniques

3.1 Choix techniques

Le contexte fonctionnel du projet (surveillance web asynchrone, extraction HTML et gestion de files de tâches) impose des contraintes techniques précises. L'organisme d'accueil a défini un ensemble de frameworks à utiliser. Nous présentons ci-dessous chacun de ces choix et les raisons qui les justifient dans ce contexte.

Framework Next.js (Front-end)

Next.js 16 est un framework React développé et maintenu par Vercel. Il étend les capacités de React en offrant un système de routage basé sur le système de fichiers, le rendu côté serveur (*Server-Side Rendering*), la génération statique, ainsi qu'une architecture orientée composants serveur (*React Server Components*).

Ses principaux atouts sont les suivants :

- ▷ **Architecture hybride (serveur/client) :** Next.js distingue les composants serveur des composants client interactifs, ce qui réduit la taille du bundle JavaScript, c'est important pour une interface avec plusieurs vues de données temps réel.
- ▷ **Routage structuré et layouts partagés :** Le système App Router permet de définir des layouts persistants entre les pages, qui offre une navigation sans rechargements.
- ▷ **TypeScript natif :** Next.js offre une intégration TypeScript, qui est un sur-ensemble de JavaScript qui se compile en JavaScript pur. Il est devenu un standard dans le web moderne grâce a sa détection précoce des erreurs, cohérence des types et interoperabilité..

Framework NestJS (Back-end)

NestJS est un framework Node.js progressif, écrit en TypeScript, inspiré de l'architecture modulaire d'Angular. Il repose sur l'injection de dépendances et les principes SOLID, ce qui en fait un choix adapté à la construction d'APIs maintenables à forte logique métier.

Ses principaux atouts sont les suivants :

- ▷ **Architecture modulaire :** NestJS organise le code en modules indépendants, ce qui facilite l'isolation des responsabilités et la testabilité de chaque composant.
- ▷ **Injection de dépendances :** Le conteneur intégré permet de déclarer les services et leurs dépendances de manière déclarative, en conformité avec le principe d'inversion de dépendances (DIP).
- ▷ **Intégration native de BullMQ :** Le module `@nestjs/bullmq` permet de définir des processeurs de queue sous forme de services injectables, cohérents avec le reste de l'architecture.
- ▷ **Écosystème TypeScript unifié :** Le partage du langage TypeScript entre front-end et back-end facilite la définition de types communs et réduit les erreurs aux points d'intégration.

ORM Prisma et base de données PostgreSQL

Prisma est un ORM pour Node.js et TypeScript qui génère automatiquement un client typé à partir d'un schéma relationnel. PostgreSQL, système de gestion de bases de données

relationnelles open-source, est retenu pour sa robustesse et son support des types JSON natifs.

Leurs atouts dans ce projet sont les suivants :

- ▷ **Typage TypeScript :** Le client Prisma expose des types TypeScript précis pour chaque modèle et requête.
- ▷ **Migrations versionnées :** Prisma Migrate génère des migrations SQL incrémentales.

File de tâches: BullMQ et Redis comme base de données

BullMQ est une bibliothèque de gestion de files de tâches pour Node.js, construite sur Redis, qu'elle utilise comme couche de persistance pour stocker l'état des files et les métadonnées de chaque job.

BullMQ offre :

- ▷ **Découplage :** BullMQ permet d'exécuter des tâches de manière asynchrone, sans bloquer le serveur HTTP.
- ▷ **Planification cron native :** BullMQ intègre un système de planification par expression cron pour les tâches récurrentes.
- ▷ **Résilience et rejeu :** En cas d'échec, BullMQ conserve les métadonnées d'exécution et peut rejouer automatiquement une tâche selon une politique configurable.

Extraction : Cheerio et Puppeteer

Deux bibliothèques d'extraction ont été orientée par la spécification fonctionnelle, répondant à des contextes différents.

Cheerio est une bibliothèque d'analyse HTML légère pour Node.js, inspirée de l'API jQuery. Elle opère directement sur le HTML statique sans instancier de navigateur, ce qui la rend rapide et excellente pour les sites web statiques.

Puppeteer instancie un navigateur Chromium, permettant d'exécuter le JavaScript de la page cible et d'accéder au DOM tel qu'il est rendu dans un vrai navigateur. Ceci est important pour les sites web qui utilise le Client-side Rendering.

4 Analyse et spécification des besoins non fonctionnels

En complément des besoins fonctionnels, la plateforme doit satisfaire un ensemble d'exigences qui conditionnent sa robustesse, sa sécurité et sa fiabilité à long terme.

- **Sécurité :**

- ✓ L'accès à la plateforme doit être protégé par un mécanisme d'authentification incluant une vérification d'e-mail.
- ✓ Les données sensibles des utilisateurs doivent être chiffrées en transit et au repos.
- ✓ Chaque ressource (URL-cible, point de données, job) est strictement isolée par utilisateur ; toute tentative d'accès non autorisé est rejetée par le système.

- **Fiabilité :**
  - ✓ L'exécution des jobs doit être gérée de manière asynchrone et découplée du reste de l'application, de sorte qu'un échec isolé n'affecte pas la disponibilité générale de la plateforme.
  - ✓ En cas d'échec d'une exécution, les informations d'erreur doivent être persistés et l'utilisateur doit être notifié avec detail.

- **Ergonomie :**
  - ✓ Les actions de configuration doivent être regroupées de manière cohérente, afin de limiter les changements de contexte pour l'utilisateur.
  - ✓ Toute opération irréversible doit être précédée d'une confirmation explicite de la part de l'utilisateur.
  - ✓ L'interface doit offrir des mécanismes de filtrage et de recherche sur les listes de données.

5 Conclusion

Ce chapitre a établi le périmètre fonctionnel et technique de la plateforme. Les besoins identifiés couvrent l'ensemble du cycle de vie d'un job de surveillance, de sa création jusqu'à l'analyse de ses résultats. Les choix technologiques retenus répondent aux contraintes d'une architecture asynchrone, qui gère l'exécution de tâches d'extraction web. Le chapitre suivant aborde la conception logicielle de ce projet.

Chapitre 3

Étude conceptuelle

1 Introduction

Comme le disait Linus Torvalds : « Les mauvais programmeurs se soucient du code. Les bons programmeurs se soucient des structures de données et de leurs relations. »
Dans ce chapitre, nous allons conceptualiser nos « structures de données et leurs relations »; la base de données, l'architecture globale de notre système, et la conception logicielle.

2 Architecture globale du système

Notre système repose sur une architecture trois-tiers, de loin la plus fameuse en développement web aujourd'hui. Cette architecture divise le système en trois couches principales :

– La couche présentation, responsable de la partie visuelle et interactive, que les utilisateurs finaux voient et expérimentent.
– La couche métier, qui héberge la logique métier.
– La couche base de données, responsable du stockage, de l'organisation et de la récupération des données.

La figure 3.1 illustre l'architecture globale, où chaque couche regroupe un ensemble de sous-couches, elles-mêmes divisées en modules. Ensuite, nous décortiquerons chacune de ces couches. Dans le chapitre suivant, nous présenterons les technologies utilisées pour leur mise en œuvre.

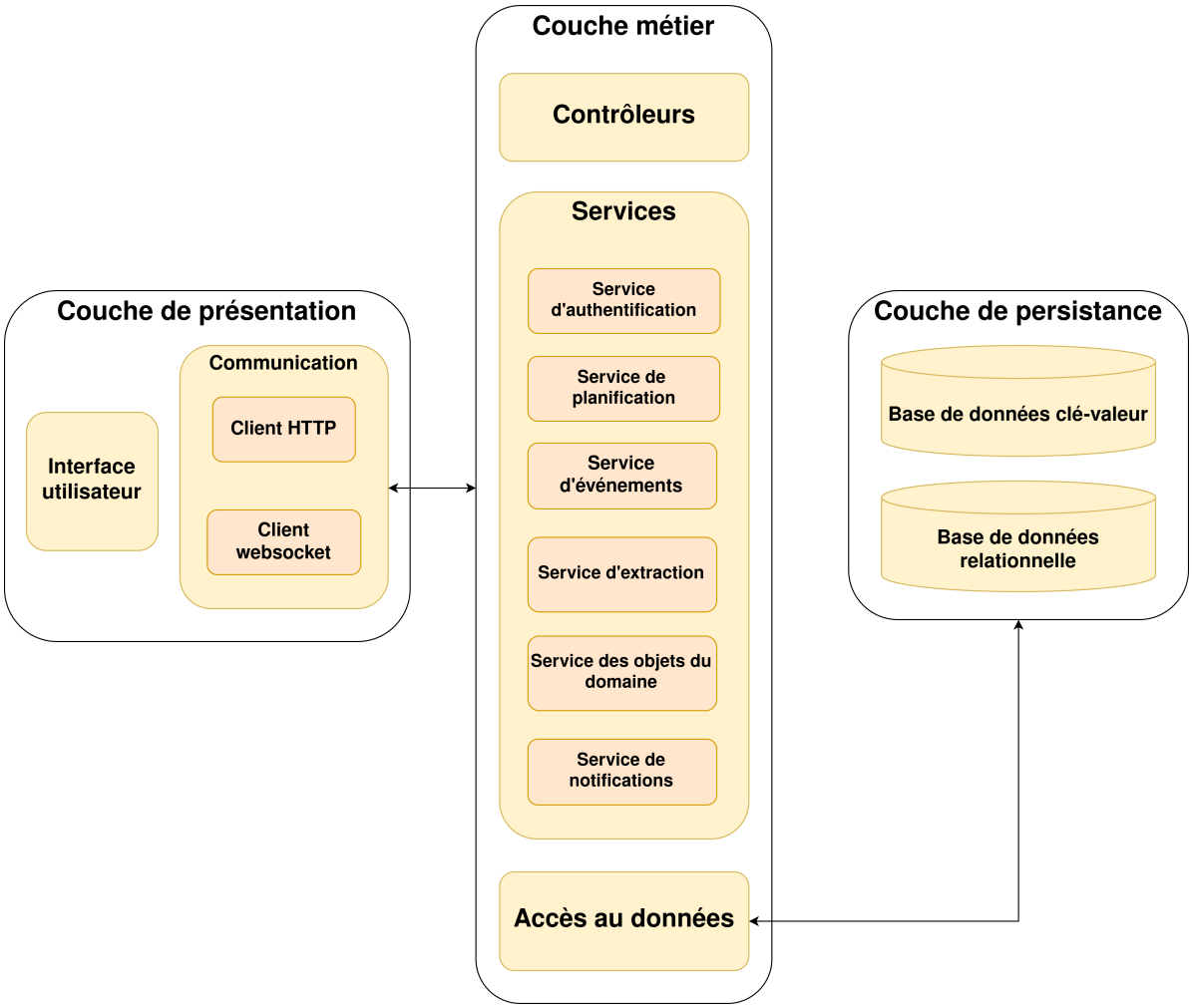


Figure 3.1. *Architecture globale du système*

Couche de présentation

La couche de présentation est une application web développée avec *Next.js*. Elle est responsable de l'affichage des données et de la collecte des interactions utilisateur, et la communication avec la couche métier.

La partie communication gère l'envoi et la récupération, des requêtes HTTP, et la communication en temps réel via les notifications. La partie interface graphique repose sur des composants web réactifs.

Couche métier

La couche métier est une API REST développée avec *NestJS*, organisée en trois sous-couches. Une couche contrôleur pour exposer les endpoints REST et de déléguer le traitement aux services sous-jacents, une couche service, qui encapsule la logique métier, et une couche d'accès aux données.

Les parties les plus significatifs de la **couche service** sont les suivants :

– **Service d'authentification:** Creation de compte, authentification, connexion et déconnexion des utilisateurs dans la plateforme.
– **Service de planification:** Enfilement et défilement des tâches asynchrones dans la couche persistance (précisemment la base de données clé-valeur).
– **Service des événements:** Création et diffusion des événements entre les différentes parties du système.
– **Service d'extraction:** Exécution des tâches (scripts) d'extraction des données des pages webs.
– **Service de notifications:** Envoie des notifications vers la couche de présentation, ou vers la boîte e-mail du client.
– **Service des objets du domaine :** Représentation orientée objet des entités métier, qui regroupe leurs données et les opérations associées.

Couche de persistance

La couche de persistance repose sur deux supports de persistance; un système de gestion de base de données relationnelle, pour le stockage et la récupération des données structuré. une base de données clé-valeur pour la gestion de la file d'attente et les tâches asynchrones.

3 Conception de la base de données

Tout programme informatique, par définition, manipule des données. Il convient donc de définir rigoureusement la structure de leurs données avant toute implémentation. Dans cette partie, nous présentons la structure de la base de données : un dictionnaire des entités métier, suivi du modèle entité-association, puis le schéma relationnel qui en découle.

3.1 Dictionnaire des entités

Le tableau 3.1 présente les entités métier identifiées pour notre application. Pour chaque entité, nous donnons une courte description et l'ensemble de ses attributs.

Table 3.1. *Tableau descriptif des entités métier de la plateforme*

Entité	**Description**	**Attributs**
User	Représente un utilisateur inscrit sur la plateforme.	*id* : identifiant unique de l'utilisateur. *email* : adresse électronique (identifiant de connexion). *name* : nom d'affichage. *password* : mot de passe. *emailVerified* : indique si l'adresse e-mail a été vérifiée. *notifyByEmail* : préférence de notification par e-mail. *createdAt, updatedAt* : horodatages de création et de mise à jour.
TargetUrl	Représente une URL cible que l'utilisateur souhaite surveiller.	*targetUrlId* : identifiant unique. *url* : adresse complète de la page surveillée. *baseUrl* : nom d'hôte extrait de l'URL (ex. : `example.com`). *name* : titre de la page. *status* : état de l'URL cible (`ACTIVE`, `INACTIVE`). *createdAt, updatedAt* : horodatages.
Datapoint	Représente un point de données défini sur une URL cible, correspondant à un ou plusieurs éléments HTML à surveiller.	*datapointId* : identifiant unique. *name* : nom attribué au point de données. *path* : sélecteur(s) CSS ciblant le ou les éléments à extraire. *fieldNames* : noms des champs d'extraction définis par l'utilisateur (sérialisés en JSON). *createdAt, updatedAt* : horodatages.

Entité	**Description**	**Attributs**
Job	Représente un job de surveillance planifié, attaché à un point de données.	*jobId* : identifiant unique. *status* : état du job (`ACTIVE`, `PAUSED`). *cron* : expression cron définissant la planification récurrente. *scheduleStart* : date de démarrage pour une exécution planifié. *extractorType* : mode d'extraction utilisé (`BASIC`, `SMART`). *outputFormat* : format de sortie des résultats (`JSON`, `CSV`, `TXT`, `MD`). *notifyFinish, notifyDiff, notifyFail* : préférences de notification. *createdAt, updatedAt* : horodatages.
Execution	Représente une instance d'exécution d'un job.	*executionId* : identifiant unique. *status* : état de l'exécution (`RUNNING`, `DONE`, `FAILED`). *startedAt* : date et heure de démarrage de l'exécution. *finishedAt* : date et heure de fin de l'exécution. *createdAt* : horodatage de création.
Result	Représente le résultat extrait lors d'une exécution, stocké dans un format choisi par l'utilisateur.	*resultId* : identifiant unique. *definition* : contenu extrait, stocké au format (`JSON`, `CSV`, `TXT` ou `MD`) *date* : date d'enregistrement du résultat.
Log	Représente une entrée de journal associée à une exécution.	*logId* : identifiant unique. *level* : niveau d'importance du message (`INFO`, `WARN`, `ERROR`, `DEBUG`). *message* : contenu du message du log. *date* : date d'enregistrement.

Entité	Description	Attributs
Notification	Représente une notification générée à la suite d'un événement système (fin d'exécution, différence détectée, échec).	*notificationId* : identifiant unique. *type* : nature de la notification (`EXECUTION_DONE`, `EXECUTION_FAILED`, `EXECUTION_DIFF`). *title, body* : titre et contenu de la notification. *read* : indique si la notification a été lue. *createdAt* : horodatage de création.

3.2 Modèle entité-association

Nous avons modélisé notre base de données à travers ce modèle entité-association 3.2

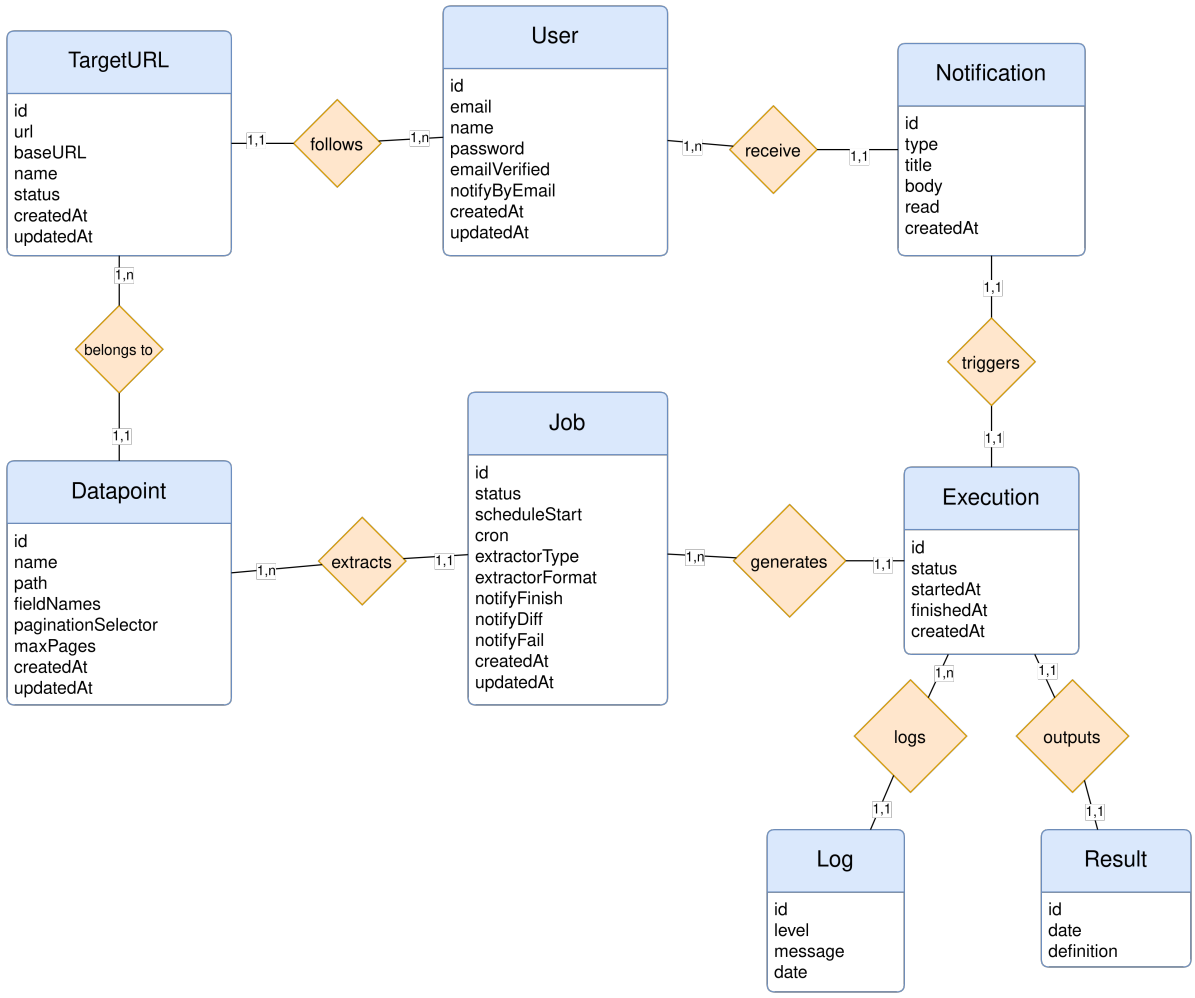


Figure 3.2. *Modèle entité-association de la base de données*

3.3 Schéma relationnel de la base de données :

Nous dérivons le schéma relationnel suivant à partir du modèle entité-association présenté précédemment. Les clés primaires sont soulignées, les clés étrangères sont précédées du symbole #.

Users(userId, email, name, password, emailVerified, provider, image, role, notifyByEmail, createdAt, updatedAt)

Target_urls(targetUrlId, url, baseUrl, name, status, createdAt, updatedAt, #userId)

Datapoints(datapointId, name, path, fieldNames, createdAt, updatedAt, #targetUrlId)

Jobs(jobId, status, definition, cron, scheduleStart, extractorType, outputFormat, notifyOnFinish, notifyOnDiff, notifyOnFail, createdAt, updatedAt, #datapointId)

Executions(executionId, status, startedAt, finishedAt, createdAt, #jobId)

Results(resultId, definition, date, #executionId)

Logs(logId, level, message, date, #executionId)

Notifications(notificationId, type, title, body, read, createdAt, #userId, #executionId)

4 Conception logicielle

Cette partie présente la conception logicielle en se basant sur le langage UML. Nous y aborderons deux aspects : la vue statique du système, illustrée par des diagrammes de classes, et la vue dynamique, représentée par des diagrammes de séquence.

4.1 Diagramme de classes

Nous organisons les diagrammes de classes en deux niveaux : les classes du domaine métier générées par **Prisma**, et les classes de la couche métier qui encapsulent la logique applicative.

Classes des entités du domaine

La couche métier s'appuie sur un ensemble de classes de domaine qui constituent la représentation objet du modèle entité-association présenté précédemment. Ces classes sont génèrées automatiquement par l'outil **Prisma**, elles encapsulent les données métier que les services manipulent tout au long du traitement. La figure 3.3 présente ces classes de domaine.

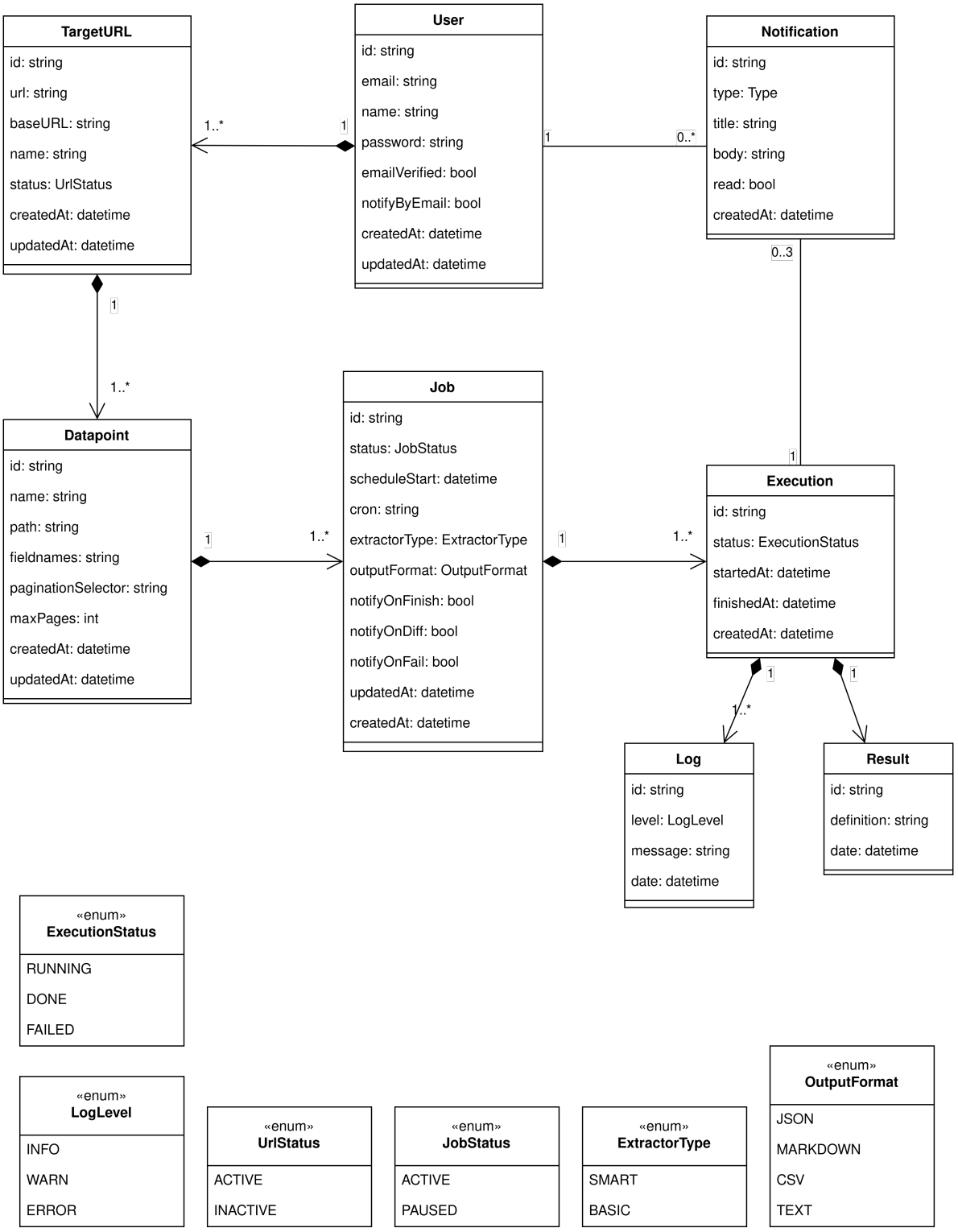


Figure 3.3. *Diagramme de classes des entités du domaine*

Classes de la couche métier

La figure 3.6 présente les classes constituant la couche métier. Cette couche concentre la logique applicative du système. Les services manipulent les données métier à travers les

classes générées par **Prisma**, puis utilisent **PrismaService** pour lire, créer, modifier ou supprimer ces données dans la base.

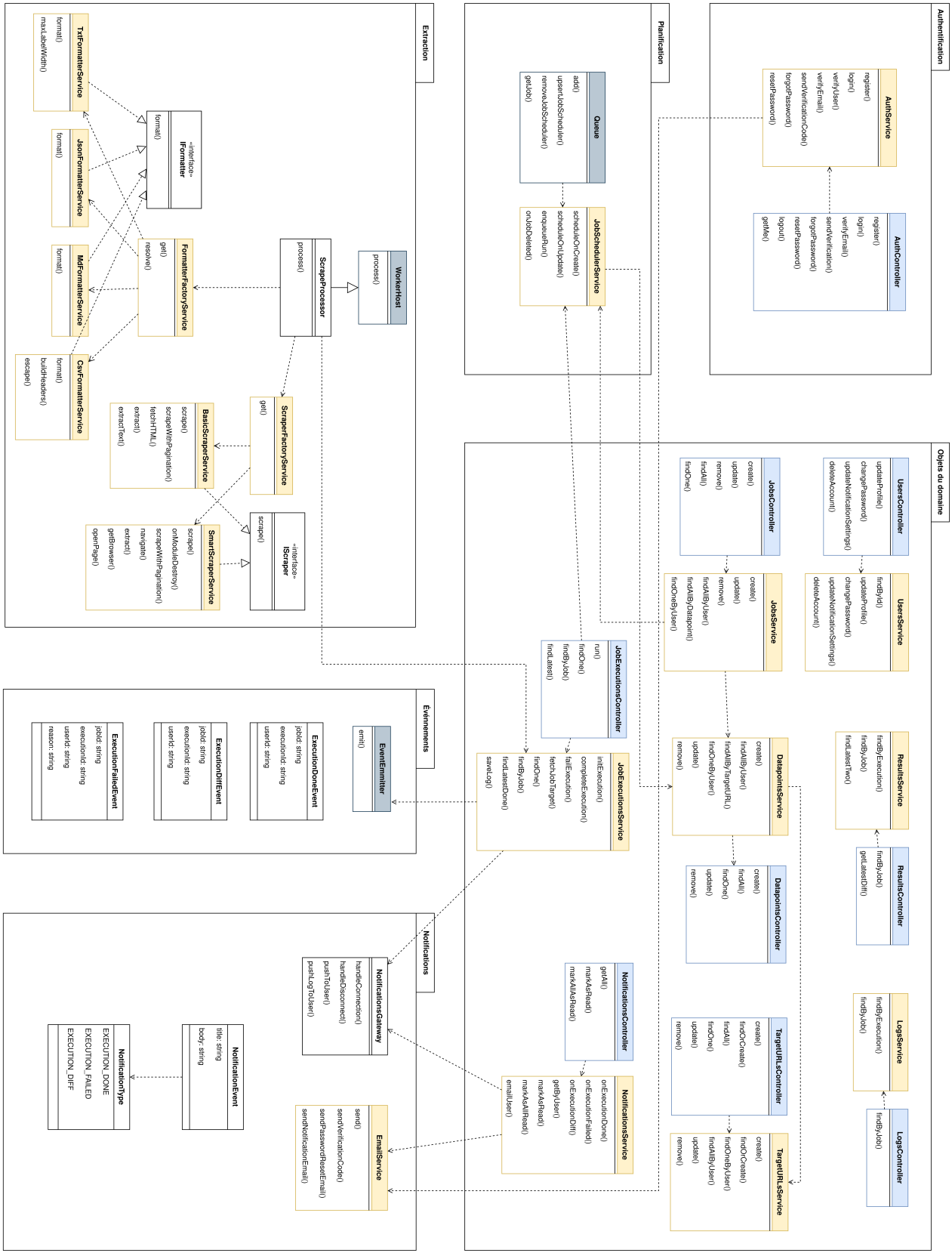


Figure 3.4. *Diagramme de classes de la couche métier*

Le tableau 3.2 décrit le rôle de chaque classe de cette couche

Table 3.2. *Rôle des classes de la couche métier*

Classe	**Rôle**
`AuthService`	Gère l'inscription, la connexion et la vérification des utilisateurs.
`AuthController`	Contrôleur de l'authentification.
`UsersController`	Contrôleur des utilisateurs.
`UsersService`	Service des utilisateurs.
`JobsController`	Contrôleur des jobs.
`JobsService`	Service des jobs.
`ResultsService`	Service des résultats.
`ResultsController`	Contrôleur des résultats.
`LogsService`	Service des logs.
`LogsController`	Contrôleur des logs.
`DatapointsService`	Service des points de données.
`DatapointsController`	Contrôleur des points de données.
`TargetURLsController`	Contrôleur des URLs cibles.
`TargetURLsService`	Service des URLs cibles.
`JobExecutionsController`	Contrôleur des exécutions.
`JobExecutionsService`	Service des exécutions.
`NotificationsController`	Contrôleur des notifications.
`NotificationsService`	Service des notifications.
`Queue`	Gère la file d'attente des tâches à exécuter.
`JobSchedulerService`	Soumet les jobs et leur planification à la file d'attente.
`WorkerHost`	Point d'entrée du worker qui consomme les tâches de la file d'attente.

Classe	**Rôle**
`ScrapeProcessor`	Orchestre le déroulement d'une exécution d'extraction.
`IFormatter`	Interface commune à tous les formateurs de données.
`FormatterFactoryService`	Instancie le formateur correspondant au format de sortie demandé.
`TxtFormatterService`	Formate les données extraites en texte brut.
`JsonFormatterService`	Formate les données extraites en JSON.
`MdFormatterService`	Formate les données extraites en Markdown.
`CsvFormatterService`	Formate les données extraites en CSV.
`ScraperFactoryService`	Instancie la stratégie d'extraction appropriée.
`IScraper`	Interface commune à toutes les stratégies d'extraction.
`BasicScraperService`	Extrait le contenu statique d'une page via Cheerio.
`SmartScraperService`	Extrait le contenu dynamique d'une page via Puppeteer.
`EventEmitter`	Émet les événements applicatifs vers les abonnés.
`ExecutionDoneEvent`	Représente l'événement déclenché à la fin d'une exécution réussie.
`ExecutionDiffEvent`	Représente l'événement déclenché lors de la détection d'une différence.
`ExecutionFailedEvent`	Représente l'événement déclenché lors de l'échec d'une exécution.
`NotificationsGateway`	Serveur WebSocket.
`EmailService`	Envoie les notifications par courrier électronique.
`NotificationEvent`	Représente le contenu d'une notification.
`NotificationType`	Énumère les types de notifications possibles.

Quelques choix de conception notables

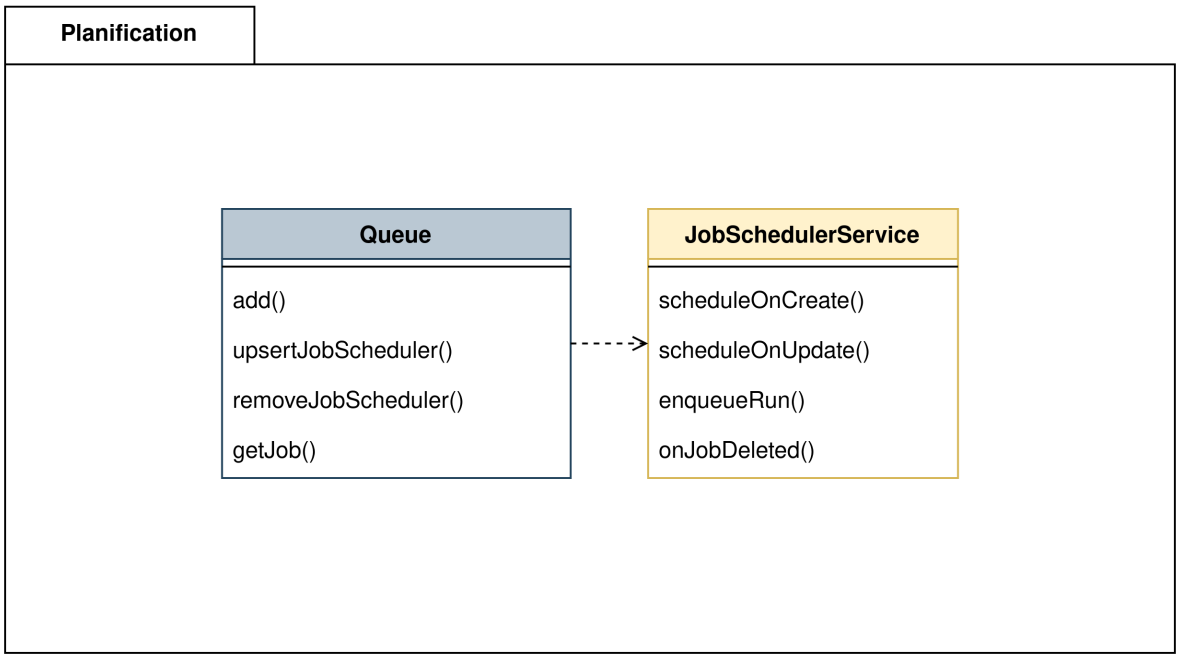


Figure 3.5. *Diagramme de classes du service de planification*

À première vue, `JobSchedulerService` semble totalement inutile ; tout aurait pu être délégué à `JobExecutionsService`, puisque `JobSchedulerService` ne fait que manipuler les exécutions. Cependant, en considérant l'étendue des operations gérées par la classe de service d'exécution, il est apparu nécessaire de séparer la logique de planification dans une autre classe, qui interagit directement avec la file d'attente, ce qui garanti la modularité et la maintenabilité du système, et de respecte le principe de responsabilité unique.

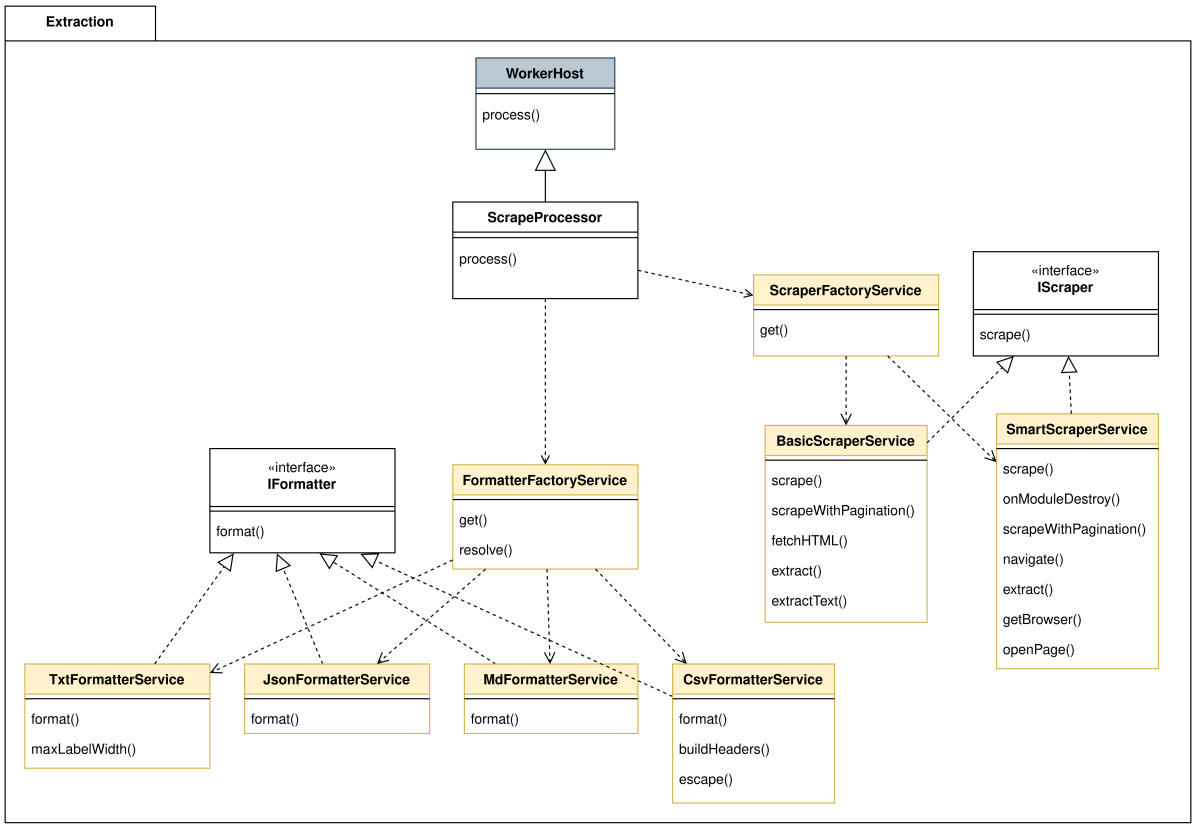


Figure 3.6. *Diagramme de classes du service d'extraction*

L'extraction du contenu web, comme nous l'avons vu au premier chapitre, peut être réalisée de nombreuses manières, voire avec différentes technologies. Nous ne pouvons pas limiter le système à une seule stratégie. C'est là ou le patron de conception « Strategy » s'avère utile. Il s'agit d'un « modèle comportemental qui définit une famille d'algorithmes, les encapsule et les rend interchangeables » [3]. Les services `BasicScraperService` et `SmartScraperService` sont interchangeables, mais ils encapsulent des algorithmes différents et constituent donc des stratégies distinctes.

Un autre patron de conception implémenté deux fois ici est le patron « Abstract Factory » : étant donné le grand nombre de formateurs (`TxtFormatter`, `CsvFormatter`...), et la possibilité de l'augmentation des stratégies d'extraction, il était logique, d'un point de vue conceptuel, de déléguer la création d'objets à une classe distincte, plutôt que d'obliger le code client à instancier directement les objets, ce qui ajoute plus de complexité inutile au `ScrapeProcessor`.

4.2 Diagrammes de séquences

Diagramme de séquence: Créer un job de surveillance

Le diagramme de séquence ci-dessous 3.7, illustre le scénario de création d'un job de surveillance. Il suppose que l'utilisateur est déjà authentifié. La séquence débute par la

soumission du formulaire de création depuis l'interface, la persistance des entités associées au job de surveillance (point de données, URL cible, job), puis le retour de la réponse à l'utilisateur. À l'issue de la création, deux cas sont distingués : si une planification immédiate est configurée, le job est enfilé dans la file d'attente; si une planification programmé par expression CRON est choisie, le planificateur enregistre la tâche sans déclencher une exécution immédiate.

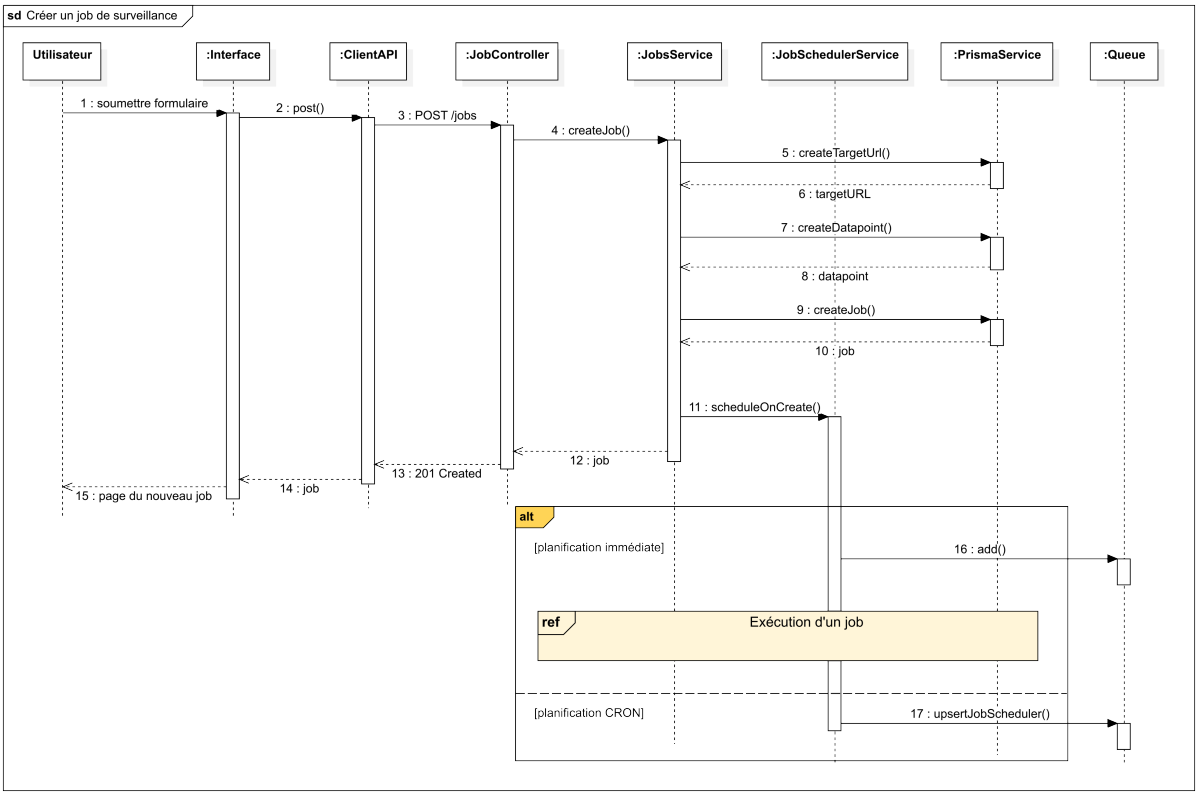


Figure 3.7. *Diagramme de séquence: Créer un job de surveillance*

Diagramme de séquence: Exécution d'un job

Le diagramme 3.8 décrit l'exécution d'un job. Le traitement se lance avec la fonction *process()*, qui est déclenchée automatiquement par BullMQ lorsqu'un job arrive ou devient disponible dans la file d'attente. Puis, le Service initialise l'exécution avec *initExecution()*. Pendant l'extraction, les logs sont enregistrés et transmis à l'interface pour suivre l'avancement.

À la fin du traitement, le Service finalise l'exécution avec *completeExecution()*, sauvegarde le résultat et vérifie s'il existe une différence avec les résultats précédents. Selon le résultat, le système déclenche une notification de succès, de changement détecté ou d'échec, puis l'envoie à l'utilisateur.

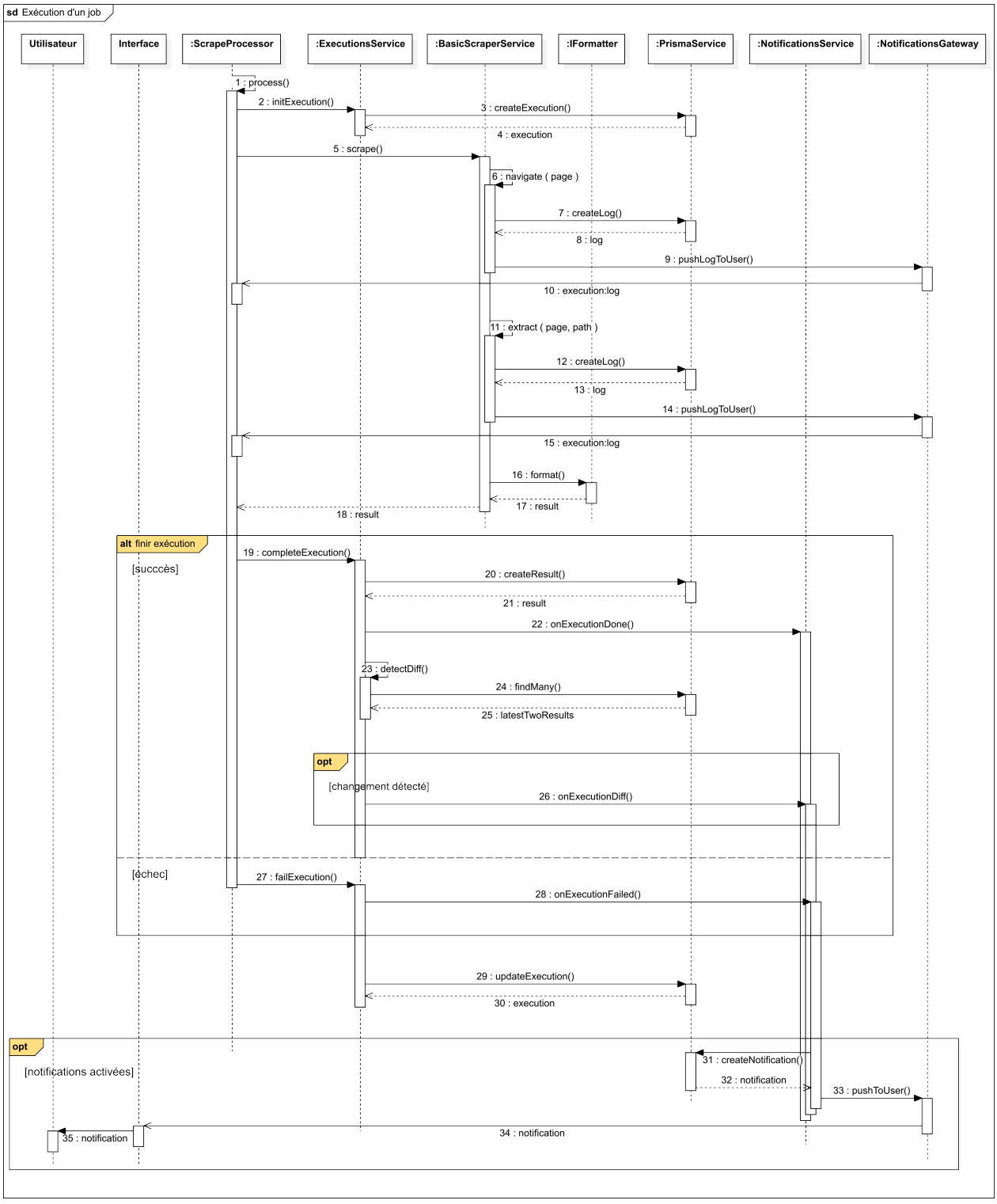


Figure 3.8. *Diagramme de séquence: Exécution d'un job*

5 Conclusion

Ce chapitre a abordé la conception du système. Nous avons défini l'architecture globale, modélisé les entités métier avec un diagramme entité-association, puis dérivé le schéma

relationnel correspondant. Nous avons ensuite présenté les diagrammes de classes de la couche métier, ainsi que les diagrammes de séquence de création et d'exécution d'un job. Le chapitre suivant présente l'implémetation de cette conception.

Chapitre 4

Implémentation

1 Introduction

Nous commençons ce chapitre par la présentation de l'environnement matériel et logiciel de développement utilisé, puis nous exposons les interfaces réalisées. Nous terminons par les tests effectués pour valider le système. Suivant le processus unifié, le développement a déroulé de manière incrémentale : chaque fonctionnalité a été intégrée progressivement, et des tests unitaires et d'intégration ont été produits (ou enrichis), et exécutés à l'issue de chaque incrément.

2 Environnement de développement

2.1 Environnement matériel

L'environnement matériel est un ordinateur portable **Lenovo IdeaPad 3 15IAU7**, dont les caractéristiques sont les suivantes :

- ▷ **Processeur :** Intel Core i5-1235U ($12^{\text{ème}}$ génération)
- ▷ **Mémoire RAM :** 16 Go
- ▷ **Stockage :** 512 Go (SSD NVMe)
- ▷ **Carte graphique :** Intel Iris Xe Graphics
- ▷ **Écran :** 15.7 pouces, résolution 1920×1080
- ▷ **Système d'exploitation :** Arch Linux x86_64

2.2 Environnement logiciel

L'environnement logiciel utilisé pour réaliser notre projet est dans le tableau suivant 4.1

Table 4.1. *Environnement logiciel du projet*

Outil / framework	**Description**
VSCodium 1.121	Éditeur de code principal utilisé pour le développement.
Firefox 151.0	Navigateur utilisé pour les tests et la validation de l'interface.
Git 2.54	Système de gestion de versions utilisé localement.
GitHub	Plateforme d'hébergement du code source.
Node.js 26.2	Environnement d'exécution utilisé pour l'application.
npm 11.14	Gestionnaire de paquets utilisé pour l'installation des dépendances.
TypeScript 5.7	Langage principal utilisé pour le développement.
NestJS 11.0	Framework backend utilisé pour la partie serveur.
Next.js 16.2	Framework frontend utilisé pour la partie client.
Tailwind CSS 4.0	Framework CSS utilisé pour le style de l'interface.
Jest 30.0	Framework de test JavaScript pour les tests unitaires.
Supertest 7.2	Bibliothèque pour les tests HTTP. Utilisé dans les tests d'intégration.
Prisma 7.4	ORM utilisé pour l'accès et la manipulation de la base de données.
PostgreSQL 18	SGBD relationnel utilisé pour le stockage des données.
Redis 8.8	Base de données clé-valeur utilisée pour les traitements asynchrones.
BullMQ 5.70	Bibliothèque utilisée pour la gestion des files de tâches.
Prisma Studio 0.511	Outil utilisé pour la visualisation de la base de données.

3 Architecture technique de l'application

L'architecture trois-tiers présentée dans le chapitre précédent (voir figure 3.1) est mise en œuvre à travers les technologies illustrées dans la figure 4.1.

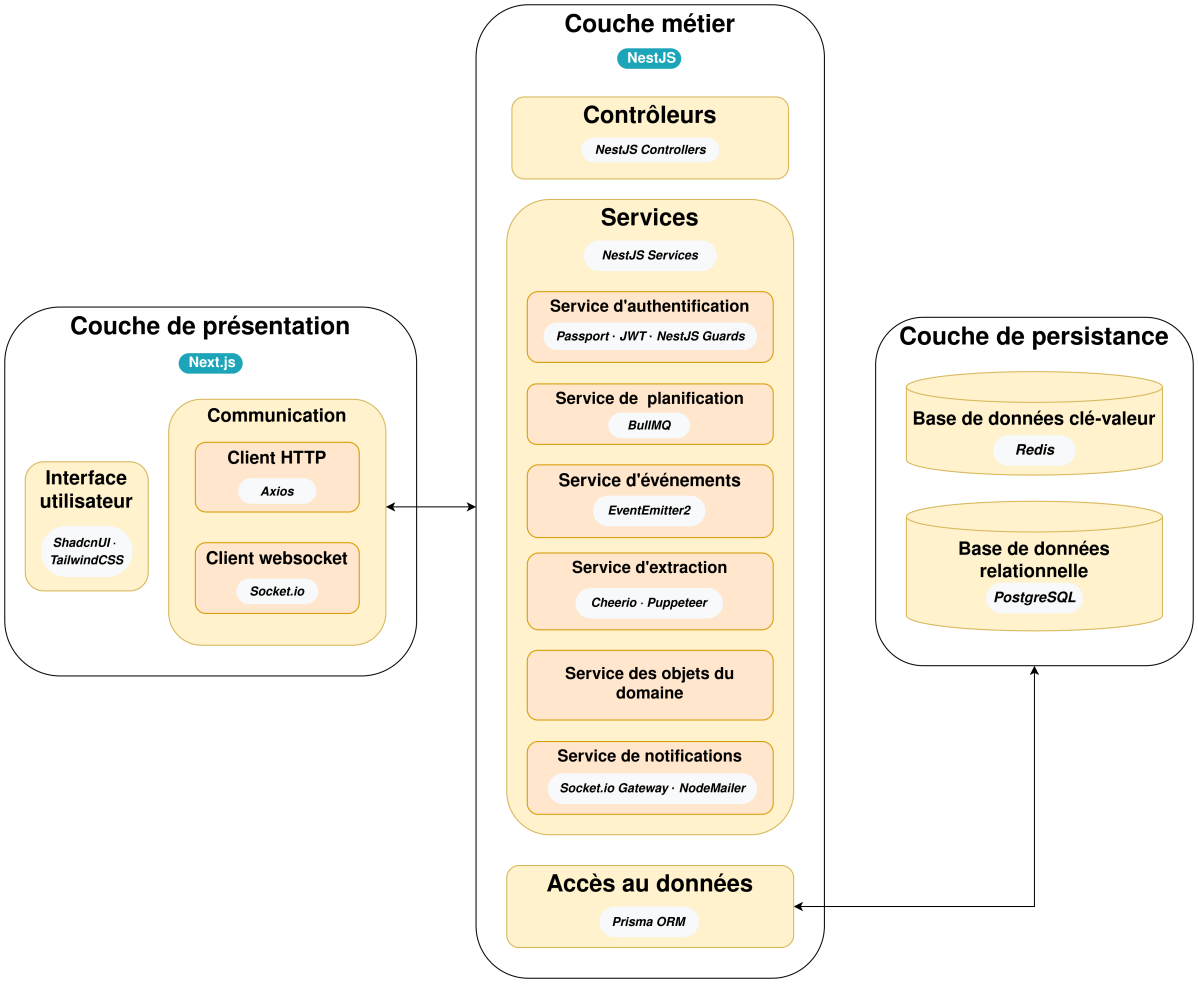


Figure 4.1. *Architecture technique de l'application*

La couche de présentation utilise **Next.js** et **React**, stylisée via *Tailwind CSS* et la bibliothèque de composants *ShadcnUI*. La communication avec le backend passe par la bibliothèque *Axios* pour les requêtes HTTP, et *Socket.io* pour la communication temps réel, notamment la diffusion des logs d'exécution et des notifications. La couche métier repose sur **NestJS**, et se compose des modules suivants :

- **Authentification :** gérée via *Passport.js*, une bibliothèque de stratégies d'authentification pour Node.js, combinée à *JWT* (JSON Web Token), un standard ouvert permettant l'échange sécurisé d'informations d'identité sous forme de jetons signés.
- **Planification :** gestion des tâches asynchrones via la bibliothèque *BullMQ*, qui manipule des files d'attentes basée sur Redis.
- **Événements internes :** diffusion d'événements entre modules via *EventEmitter2*.
- **Extraction :** scraping et analyse des pages web via *Cheerio* et *Puppeteer*.

- ▷ **Notifications :** envoi de notifications temps réel via *Socket.io Gateway* et d'e-mails via *NodeMailer*.
- ▷ **Accès aux données :** abstraction de la base de données via l'ORM *Prisma*.

La couche de persistance repose sur **PostgreSQL** pour le stockage des données relationnelles, et **Redis** pour la persistance des files d'attentes BullMQ.

4 Déploiement

Dans cette section, nous présentons l'infrastructure de déploiement du système. Nous décrivons la répartition des couches sur les différentes plateformes d'hébergement, puis nous exposons la stratégie d'intégration et de déploiement continus mise en place.

4.1 Diagramme de déploiement

La figure 4.2 présente le diagramme de déploiement de l'application, illustrant la répartition des composants sur les différentes plateformes d'hébergement.

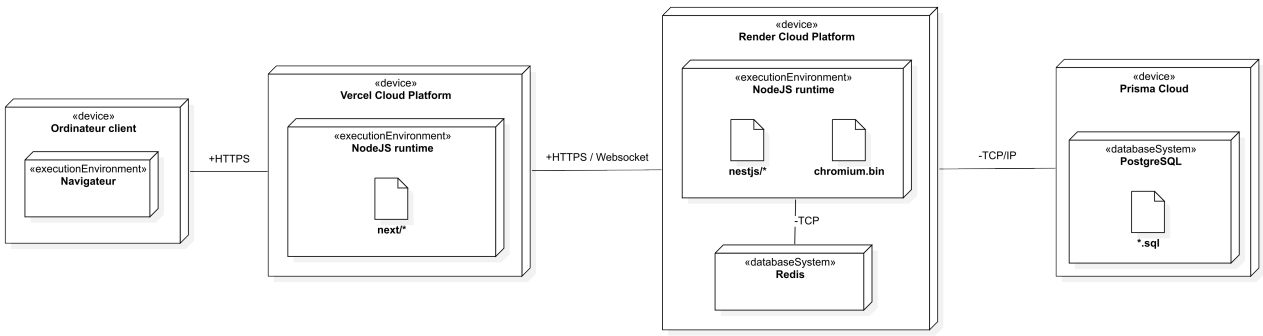


Figure 4.2. *Diagramme de déploiement de l'application*

L'infrastructure de déploiement repose sur trois plateformes distinctes. Le frontend tourne sur **Vercel**, qui assure la compilation et la distribution du build *Next.js* via un réseau de diffusion mondial (CDN). Le backend *NestJS* et la base de données *Redis* s'exécutent sur **Render** en tant que deux services indépendants sur le même compte. La base de données relationnelle réside sur **Prisma Postgres**, le service de base de données de Prisma. Les secrets applicatifs, clés JWT et identifiants SMTP, transitent par les variables d'environnement de chaque plateforme, et n'apparaissent jamais dans le dépôt source.

4.2 Justification des plateformes de déploiement choisis

4.2.1 Vercel pour la couche de présentation

Vercel est la plateforme de référence pour le déploiement des applications Next.js. Son intégration native avec le framework simplifie la configuration et réduit le temps de

mise en production. Le niveau gratuit a été retenu, le trafic étant limité en phase de développement.

4.2.2 Render pour la couche métier

Render a été retenu pour l'hébergement du backend en raison de son niveau gratuit, de sa liaison directe avec GitHub, et de sa prise en charge native des processus en arrière-plan et des tâches planifiées, requis par le moteur d'extraction.

4.3 Intégration et déploiement continus

Vercel et **Render** assurent le déploiement continu nativement, tous deux connectés au dépôt GitHub du projet. Chaque push sur la branche `main` déclenche automatiquement un pipeline de build et de déploiement sur les deux plateformes, sans intervention manuelle. Côté frontend, Vercel exécute `next build` et publie le résultat sur son CDN. Côté backend, Render exécute `npm run build`.

4.4 Quelques décisions notables lors du déploiement

4.4.1 Installation de Chrome en environnement de production

L'environnement d'exécution de Render ne dispose pas de Chrome préinstallé. Or, `SmartScraperService` en dépend pour l'extraction dynamique via Puppeteer. Le script de build suivant résout ce problème :

Listing 4.1: Script de build de la couche métier

```
#!/usr/bin/env bash
set -o errexit
npm install
npm run build
PUPPETEER_CACHE_DIR=/opt/render/.cache/puppeteer
mkdir -p $PUPPETEER_CACHE_DIR
npx puppeteer browsers install chrome
```

- `set -o errexit` : interrompt le script dès qu'une commande échoue, ce qui empêche un déploiement partiel en cas d'erreur.
- `npm install` et `npm run build` : installent les dépendances et compilent le code TypeScript en JavaScript.
- `PUPPETEER_CACHE_DIR=/opt/render/.cache/puppeteer` : redirige le cache Puppeteer vers un répertoire persistant entre les déploiements.
- `mkdir -p` : crée le répertoire s'il n'existe pas encore.

▷ `npx puppeteer browsers install chrome` : installe Chrome dans le répertoire défini, le rendant disponible dès le démarrage du serveur.

4.4.2 Architecture de communication frontend-backend

L'architecture semble simple : la couche de présentation communique avec la couche métier via HTTP. Cependant, le fait que ces couches soient déployées sur des URL différentes pose un problème de gestion des cookies d'authentification.

Problème: Lors de la connexion, l'API Render répond avec un en-tête `Set-Cookie: accessToken` lié au domaine `onrender.com`. Le navigateur stocke ce cookie sous ce domaine. Lors d'une navigation vers `/dashboard` (ou n'importe quelle page SSR), la requête est envoyée à Vercel ; le navigateur n'y joint aucun cookie, car `accessToken` n'appartient pas au domaine `vercel.app`. Le code serveur Next.js, qui lit les cookies via `cookies().get("accessToken")`, ne trouve donc rien et redirige systématiquement vers la page de connexion (figure 4.3).

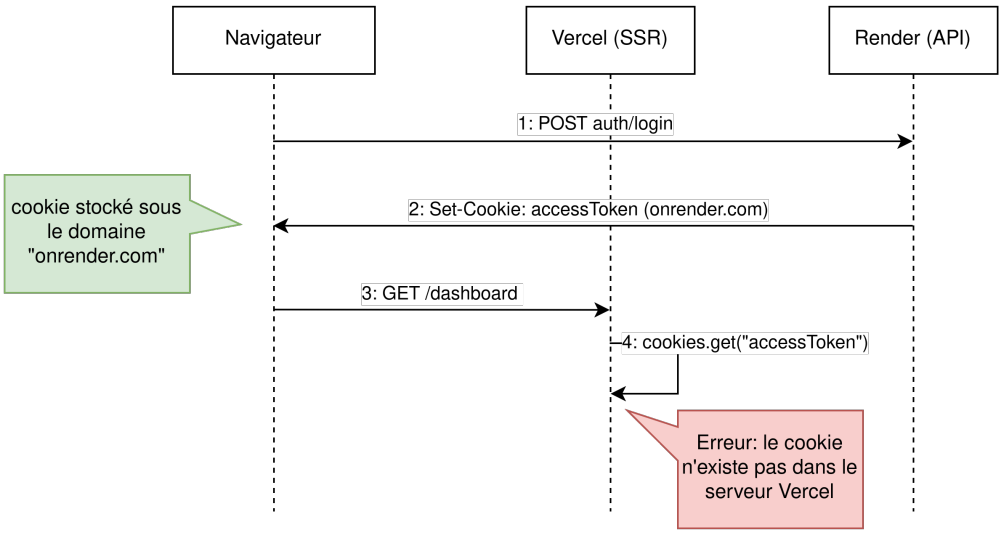


Figure 4.3. *Problème de domaine : le cookie* `accessToken`

Solution: J'ai ajouté un proxy côté client (`/proxy/...`). Les requêtes d'authentification passent par ce proxy: il contacte l'API Render, reçoit le `Set-Cookie` d'origine, supprime l'attribut `Domain=onrender.com`, puis retransmet le cookie au navigateur sous le domaine `vercel.app`. Le cookie est désormais joint automatiquement à chaque requête vers le serveur Vercel, ce qui lui permet de le lire et envoyer son contenu (accessToken) à l'API Render (figure 4.4).

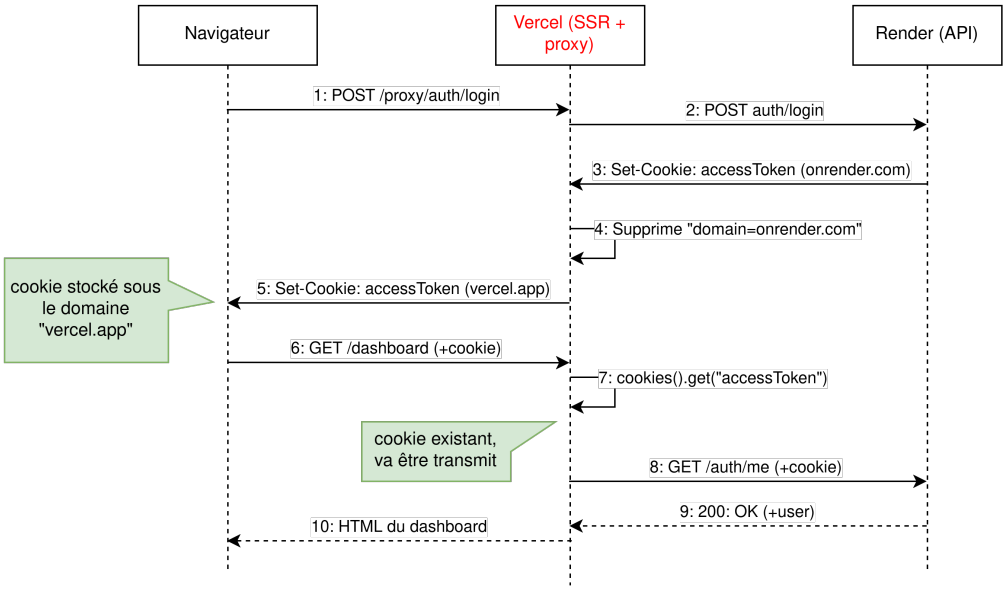


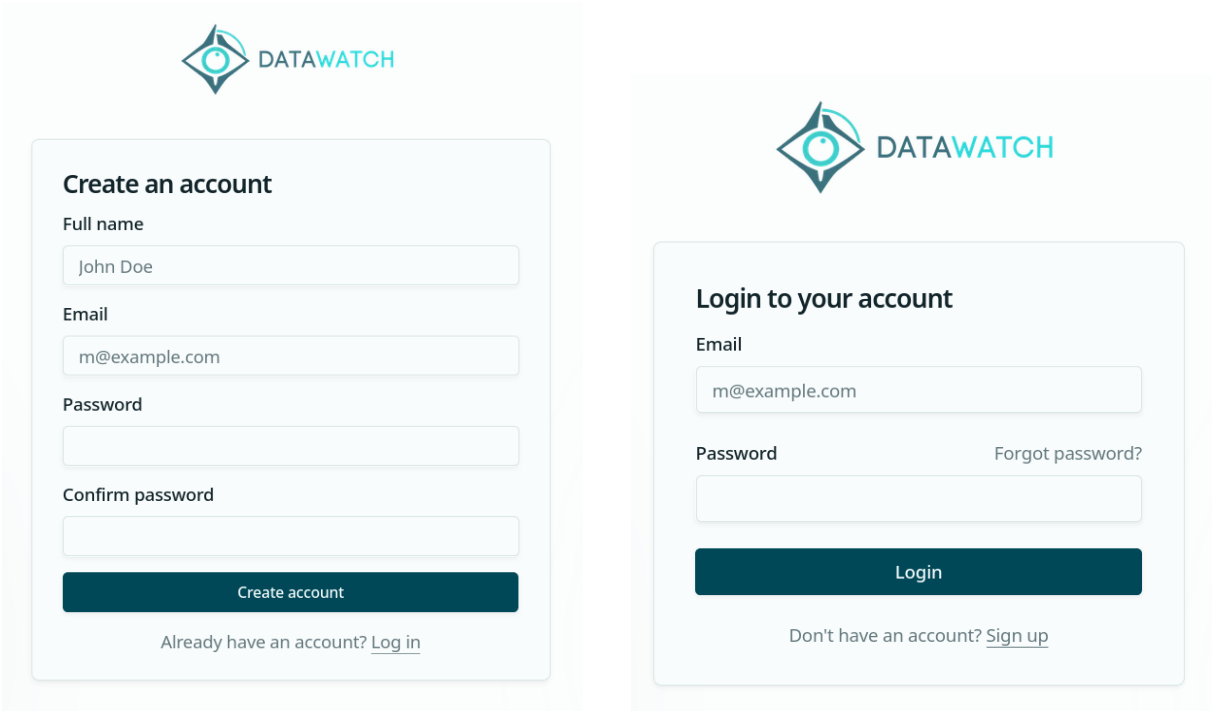
Figure 4.4. *Solution par proxy: le cookie est réémis sous* `vercel.app`

Note: Les connexions WebSocket, sont incompatible avec l'infrastructure serverless de Vercel, donc ils continuent de s'établir directement vers `onrender.com` sans passer par le proxy.

5 Présentation des interfaces

5.1 Authentification

L'interface d'inscription (figure 4.5a) permet à un nouvel utilisateur de créer un compte via un formulaire. L'interface de connexion (figure 4.5b) permet à un utilisateur existant d'accéder à la plateforme en s'authentifiant par e-mail et mot de passe.



(a) *Interface d'inscription* **(b)** *Interface de connexion*

Figure 4.5. *Interfaces d'authentification*

5.2 Tableau de bord

Le tableau de bord présente l'ensemble des cibles surveillées par l'utilisateur, pour chaque cible son statut (actif ou inactif), le nombre de datapoints associés, et la date de création. Une barre de recherche et un filtre de tri permettent de naviguer parmi les cibles. Un bouton *Build new job* donne accès directement à la création d'un nouveau job.

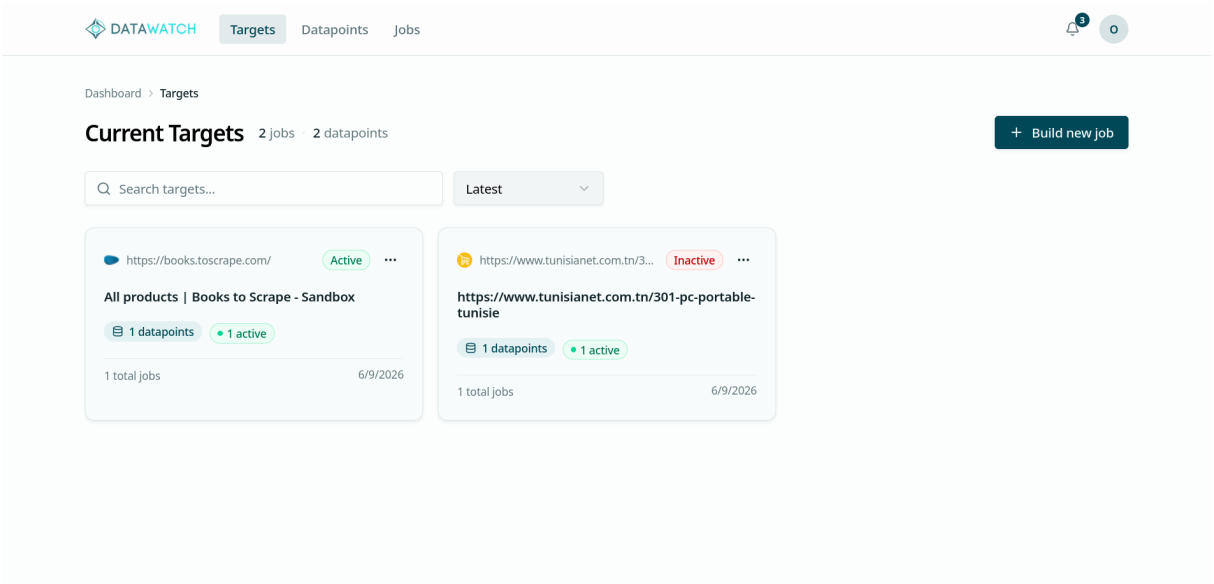


Figure 4.6. *Tableau de bord — liste des cibles surveillées*

5.3 Création d'un job de surveillance

La page de création d'un job guide l'utilisateur à travers quatre étapes : la définition du datapoint (URL cible, sélecteur CSS, champs extraits et pagination), la planification de l'exécution (ponctuelle ou récurrente), le paramétrage des notifications, et le choix du format de sortie. L'utilisateur peut sélectionner les éléments à surveiller directement sur la page cible grâce à l'outil de sélection visuelle.

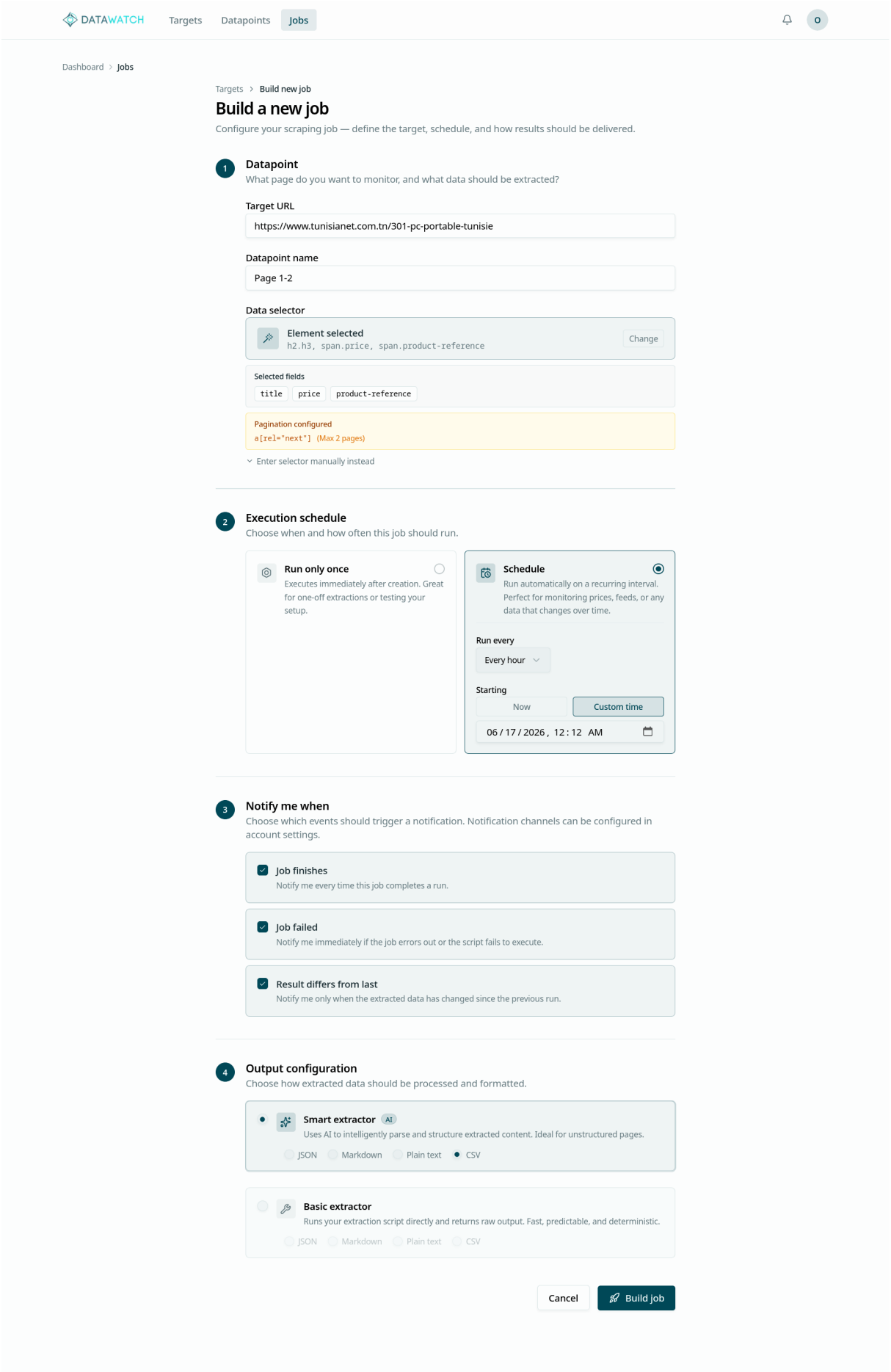


Figure 4.7. *Interface de création d'un job de surveillance*

L'outil de sélection visuelle (*element picker*) s'ouvre dans une fenêtre modale et affiche la page cible en temps réel. L'utilisateur clique sur les éléments à extraire, et le sélecteur CSS correspondant est généré automatiquement. La configuration de la pagination multi-pages est également disponible depuis cette interface.

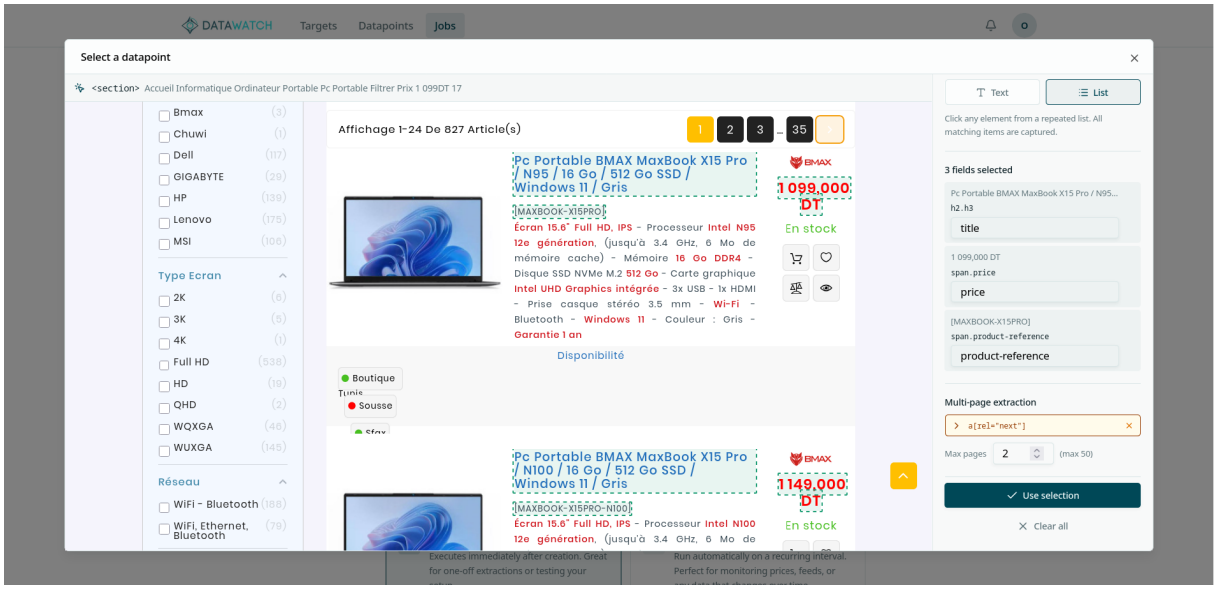


Figure 4.8. *Outil de sélection visuelle des éléments à extraire*

5.4 Gestion des jobs

La page de détail d'un job récapitule l'ensemble de sa configuration : l'URL cible, le datapoint associé, la planification, le type d'extracteur, le format de sortie et les préférences de notification. Des actions rapides permettent de lancer, mettre en pause, consulter les exécutions ou supprimer le job.

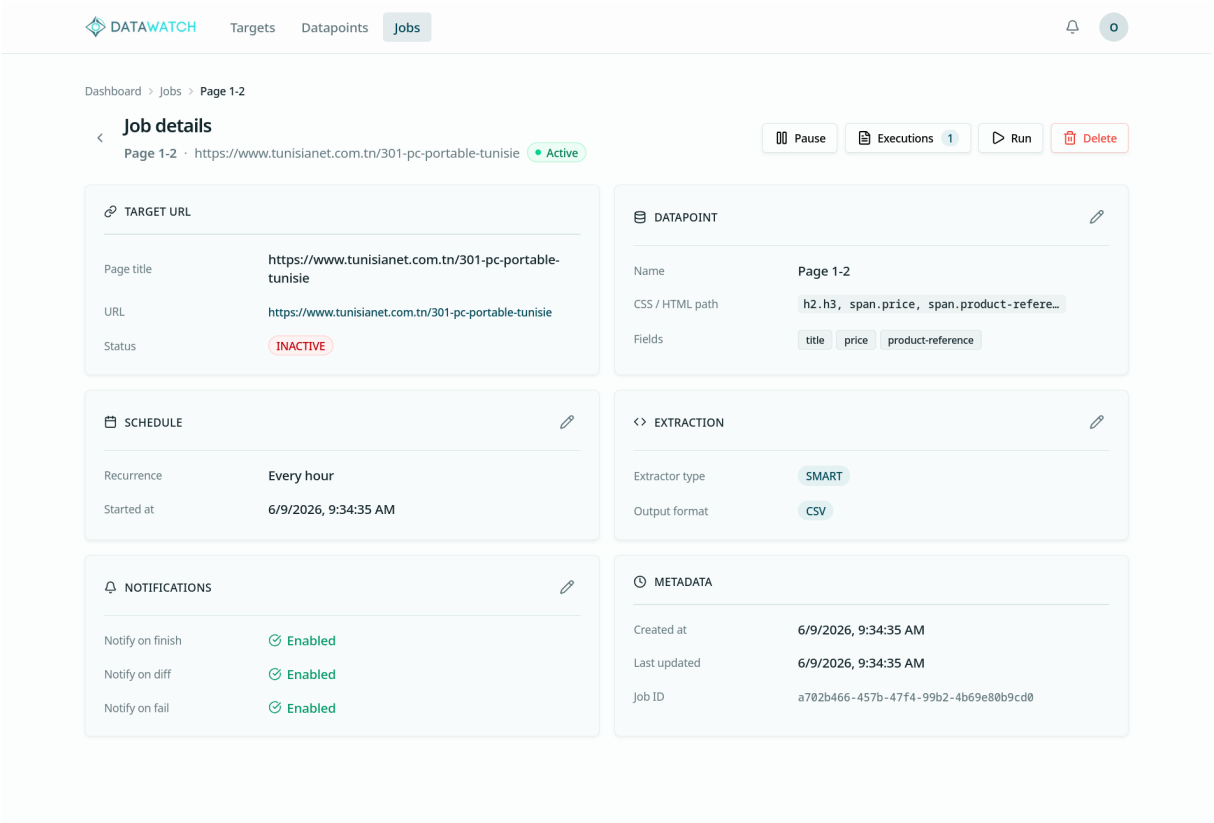


Figure 4.9. *Page de détail d'un job*

5.5 Suivi des exécutions

La page des exécutions liste l'historique des exécutions d'un job, avec le statut de chaque exécution (en cours, terminée, échouée), la date de lancement, la durée et un aperçu du dernier log enregistré.

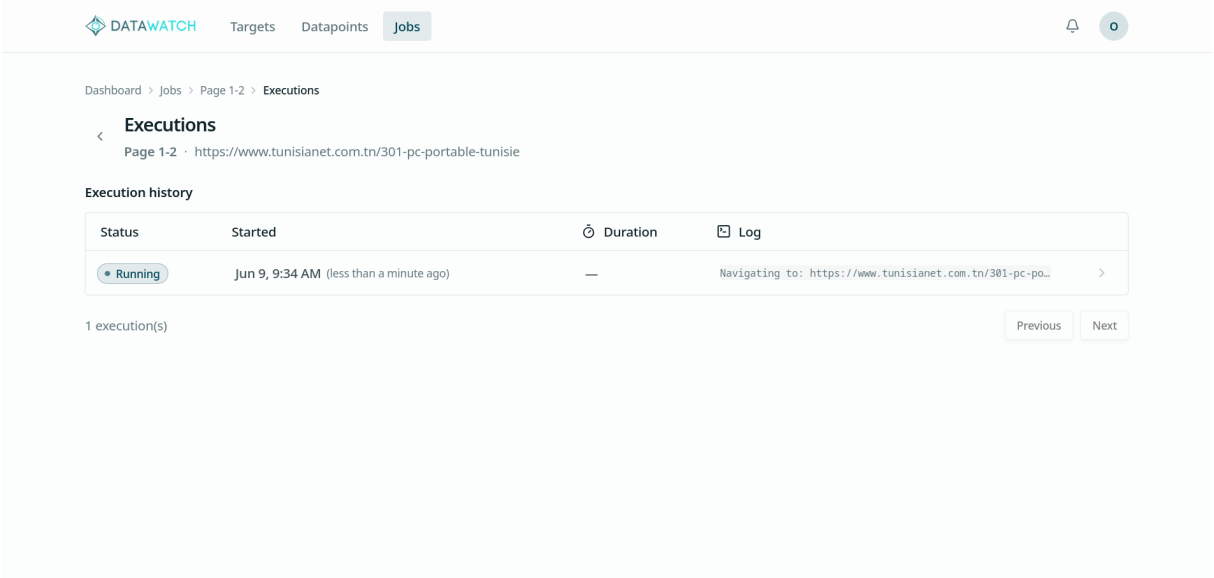


Figure 4.10. *Historique des exécutions d'un job*

La page de détail d'une exécution affiche les données extraites sous forme de tableau, ainsi que les logs d'exécution en temps réel, horodatés et classés par niveau (*INFO*). Un bouton d'export permet de télécharger les résultats.

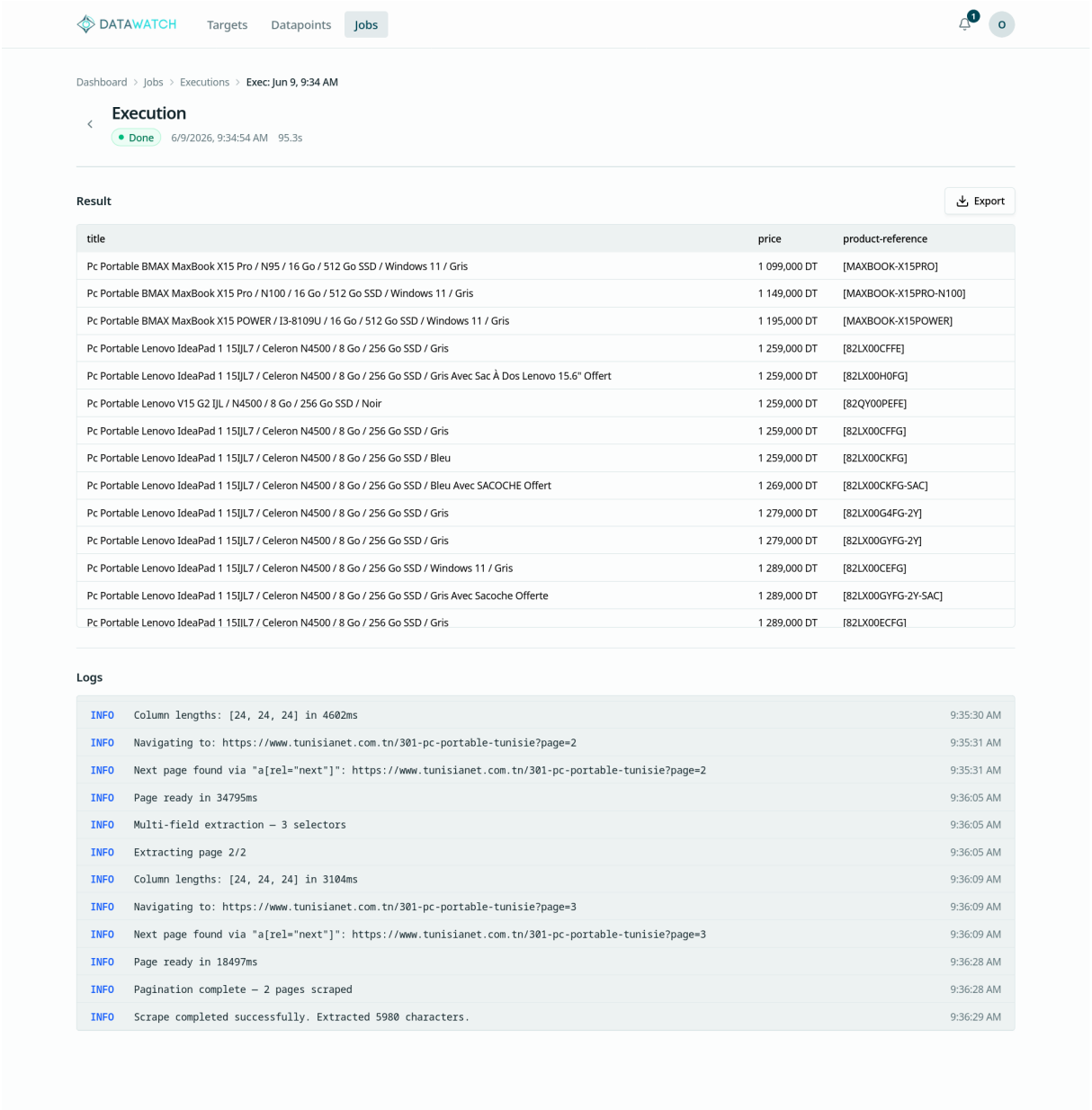


Figure 4.11. *Détail d'une exécution — résultats et logs*

5.6 Gestion des datapoints

La page des datapoints liste l'ensemble des datapoints définis par l'utilisateur, avec pour chaque entrée le nom, l'URL cible, les champs extraits et la date de création.

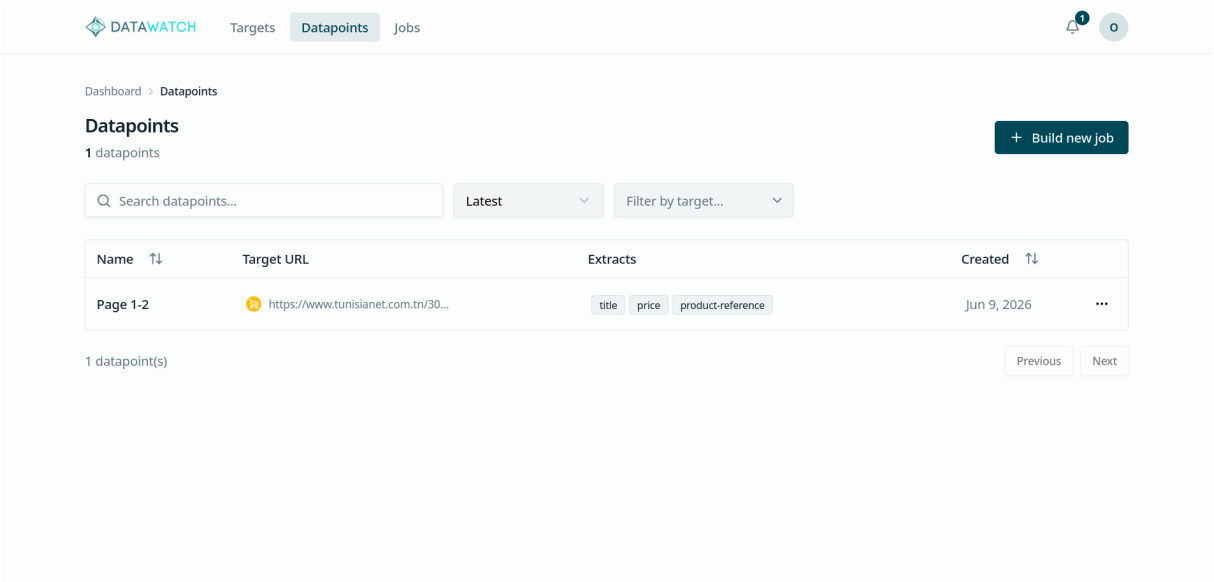


Figure 4.12. *Liste des datapoints*

La page de détail d'un datapoint affiche sa configuration complète ainsi qu'un aperçu en temps réel des données correspondant au sélecteur CSS défini, directement sur la page cible.

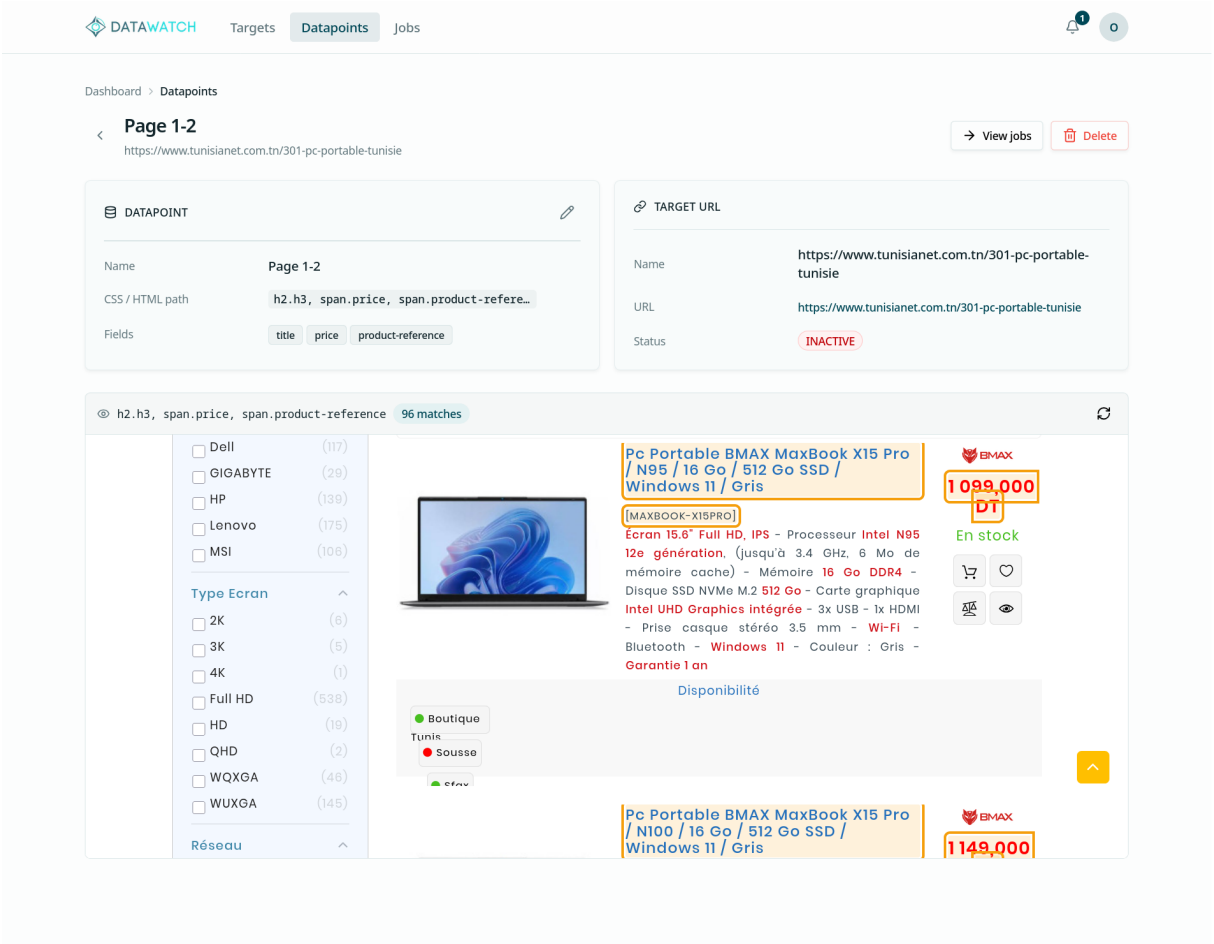


Figure 4.13. *Détail d'un datapoint avec aperçu en temps réel*

5.7 Détection des changements

Lorsqu'une différence est détectée entre deux exécutions successives, la plateforme affiche une comparaison côte à côte entre l'exécution précédente et la plus récente. Les lignes ajoutées, supprimées ou modifiées sont mises en évidence.

Figure 4.14. *Vue de comparaison — changement détecté entre deux exécutions*

6 Tests et validation

Conformément à la méthodologie 2TUP, nous avons implémenté plusieurs tests sur les modules principaux. À chaque itération, de nouveaux tests ont été ajoutés (ou améliorés) pour assurer le bon fonctionnement soit du composant individuellement (tests unitaires), soit de plusieurs composants simultanément dans un scénario donné (tests d'intégration).

6.1 Tests unitaires

Les tests unitaires, réalisés en *boîte blanche* (les cas de tests sont conçu en connaissant l'implémentation), ont pour objectif de vérifier le comportement isolé de chaque composant, indépendamment de ses dépendances externes.

Ces tests sont écrits avec **Jest** et organisés en dix suites couvrant les couches service et contrôleur de l'application. Le tableau 4.2 récapitule les suites de tests unitaires et les aspects vérifiés.

Table 4.2. *Suites de tests unitaires*

Suite de tests	**Aspects vérifiés**
`users.service.spec.ts`	Instanciation correcte du service
`auth.service.spec.ts`	Absence du mot de passe dans la réponse d'authentification
`auth.controller.spec.ts`	Fonctionnement du contrôleur d'authentification
`auth.guard.spec.ts`	Validation du guard JWT
`app.controller.spec.ts`	Fonctionnement du contrôleur principal
`job-access.service.spec.ts`	Contrôle d'accès
`jobs.service.spec.ts`	Création, modification et suppression de jobs
`job-executions.service.spec.ts`	Cycle de vie des exécutions : `RUNNING→DONE`, `RUNNING→FAILED`, émission d'événements
`basic.scraper.spec.ts`	Extraction CSS simple, multi-sélecteur et sélecteur absent
`formatters.spec.ts`	Formatage des résultats en TXT, JSON et CSV

L'exécution de l'ensemble de ces suites produit le résultat suivant :

```
Suites de tests : 10 réussies sur 10
Tests       : 22 réussis sur 22
Durée       : 1,444 s
```

6.2 Tests d'intégration

Les tests d'intégration, réalisés en boîte grise (les tests sont effectués exclusivement sur l'interface HTTP, indépendamment du contenu du code source.), ont pour objectif de valider les interactions entre les modules du système dans des conditions proches de la production. Contrairement aux tests unitaires, ils sollicitent l'application dans son ensemble : une instance NestJS réelle est démarrée, connectée à une base de données PostgreSQL locale dédiée aux tests, et les requêtes HTTP sont émises via la bibliothèque **Supertest**.

Quatre suites de tests d'intégration ont été réalisées, couvrant les ressources principales de l'API. Le tableau 4.3 présente les scénarios validés pour chaque suite.

Table 4.3. *Suites de tests d'intégration*

Fichier	**Scénarios vérifiés**
`auth.e2e-spec.ts`	Inscription d'un nouvel utilisateur, rejet des emails utilisés, rejet des payloads invalides, connexion d'un utilisateur vérifié, rejet des mauvais identifiants, accès au profil authentifié, rejet des accès non authentifiés.
`targeturls.e2e-spec.ts`	Création, lecture, modification et suppression d'une URL cible, rejet des URLs invalides, isolation entre utilisateurs, rejet des accès non authentifiés.
`datapoints.e2e-spec.ts`	Création, lecture, modification et suppression d'un datapoint, rejet de création sur une URL cible appartenant à un tiers, isolation entre utilisateurs, rejet des accès non authentifiés.
`jobs.e2e-spec.ts`	Création, lecture, modification et suppression d'un job, rejet des payloads invalides, isolation entre utilisateurs, vérification du comportement après suppression.

L'exécution de l'ensemble de ces suites produit le résultat suivant :

```
Suites de tests : 4 réussies sur 4
Tests       : 52 réussis sur 52
Durée       : 7,184 s
```

6.3 Bilan

L'ensemble de la campagne de tests totalise **74 tests répartis en 14 suites**. Afin d'évaluer la proportion du code effectivement exercée, on doit calculer la couverture de tests, sur les deux niveaux (tests unitaires + tests d'integration).

Le tableau 4.4 présente la couverture par instruction (*statements*) des modules principaux, mesurée à l'issue de l'exécution des tests unitaires et des tests d'intégration.

Table 4.4. *Couverture de tests par module*

Module	**Instructions**	**Branches**	**Fonctions**
`src/auth`	69%	54%	59%
`src/jobs`	89%	69%	89%
`src/datapoints`	95%	80%	86%
`src/target-urls`	78%	64%	81%
`src/job-executions`	69%	62%	50%
`src/scraper`	85%	61%	83%
Ensemble du projet	**70%**	**55%**	**51%**

Les modules métier principaux: `jobs`, `datapoints` et `target-urls`, atteignent une couverture par instruction supérieure à 78%.

Note: La couche de présentation est difficile à tester. Son rôle est de gérer les interactions utilisateur. Ces interactions sont difficiles à simuler automatiquement et peuvent être testées manuellement. De plus, ces tests manuels s'inscrivent plutot dans les tests d'acceptation qui ont été réalisés par l'encadrant professionnel. Les tests automatisés (unitaires et d'intégration) ont donc été concentrés sur la couche métier.

7 Conclusion

Même lorsque la conception est aboutie et qu'il ne reste que l'implémentation, les connaissances techniques ne suffisent pas. La confrontation aux contraintes réelles du déploiement peut révéler des problémes qui nous emmene à revisiter les étapes précédentes. Dans ce dernier chapitre, nous avons décrit l'environnement de développement, ainsi que l'architecture technique de l'application, puis présenté comment le système a été deployé. Nous avons ensuite présenté les interfaces principales de la plateforme, et enfin, validé le système avec des tests unitaires et des tests d'intégration.

Conclusion et perspectives

Ce projet porte sur la conception et le développement d'une plateforme d'extraction et de suivi de données web. Le contexte initial présentait le problème suivant : les données web évoluent constamment et aucune solution ne permet simultanément de : surveiller tous types de sites web, de filtrer les données bruitées et de visualiser efficacement les changements.

La méthodologie adoptée est 2TUP (Two track unified process) qui repose sur la séparation des besoins fonctionels et techniques. La première branche fonctionnelle a permis d'analyser et de définir les cas d'utilisation et les besoins des utilisateurs. La seconde branche qui est accosié aux besoins techniques, a conduit à définir clairement les technologies à utiliser. Dans la dernière étape, nous avons fusionné les deux analyses précédentes pour établir une conception logicielle, un schéma de base de données, puis nous avons procédé à l'implémentation, aux tests et au déploiement.

Le développement a abouti à la création d'une plateforme web à trois couches, déployée sur trois plateformes cloud distinctes. Bien que la séparation des responsabilités de chaque couche a considérablement facilité la maintenabilité du projet, elle a également engendré certaines difficultés. L'une des plus notables concerne le module d'authentification basé sur les cookies entre domaines différents, qui a nécessité un changement structurel de la communication entre les couches présentation et métier.

Le système a été testé et validé au moyen de 74 tests au total, combinant des tests unitaires et des tests d'intégration. Ces tests ont couvert 78 % des instructions des principaux modules de la couche métier. La couche présentation, plus complexe à tester automatiquement, a été testée manuellement par l'encadrant professionnel.

Nous sommes conscients des limitations de ce projet, notamment la couverture incomplète des tests, la gestion partielle de la détection des bots et le visualiseur de différences limité à la comparaison textuelle. Ces limitations définissent les axes d'amélioration du projet : une fonctionnalité de rotation des proxys pour contourner les mécanismes contre-bots, l'analyse sémantique des données extraites afin de catégoriser les modifications détectées, et enfin, une architecture de microservices pour le backend, afin d'isoler le service d'extraction, susceptible de devenir un goulot d'étranglement en cas d'augmentation du trafic sur la plateforme.

Bibliographie

[1] Visualping, *Competitor Price Tracking: Data from 9,705 Active Monitors*, `https://visualping.io/blog/top-tools-competitor-price-tracking`, Consulté le 27 juin 2026, 2024.

[2] P. Roques et F. Vallée, *UML 2 en action*. Eyrolles, 2004, isbn : 2212114621.

[3] E. Gamma, R. Helm, R. Johnson et J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts : Addison-Wesley, 1994.

Résumé

Ce rapport présente la conception et le développement d'une plateforme d'extraction de données web et de détection des modifications. Le projet a été réalisé dans le cadre d'un stage de fin d'études chez VISIOAD, pour l'obtention du diplôme d'ingénieur en génie logiciel.

La plateforme, destinée aux entreprises comme aux particuliers, permet de surveiller des pages web cibles, de définir les éléments à surveiller et de recevoir des alertes automatiques en cas de mise à jour de contenu.

Mots clés: Extraction de données web, suivi de changements du contenu web, planification des tâches, NestJS, Next.js, BullMQ, Redis, PostgreSQL

Abstract

This report presents the design and development of a web data-extraction and change-detection platform. The project is accomplished as a final-year internship at VISIOAD, toward completing the requirements for the Software Engineering degree.

The platform, dedicated to both enterprises and individuals, enables the user to track target web pages, define the elements to monitor, and receive automated alerts upon content changes.

Keywords: Web data extraction, change detection, job planification, NestJS, Next.js, BullMQ, Redis, PostgreSQL