Université Paris 1 Panthéon-Sorbonne

Master 2 MoSEF

Automatisation de la saisie de formulaires web

Web Scraping & API avec Python

Projet réalisé dans le cadre du cours de
Webscraping & API

Auteurs :

Juan

Shawel
Amel Cherbi

Année universitaire 2025–2026

1 Introduction

Les formulaires web sont omniprésents : inscription, réservation, saisie d'informations personnelles. Leur remplissage répétitif représente une charge importante pour les utilisateurs. Ce projet automatise cette tâche en développant une API capable de détecter et pré-remplir des formulaires de manière générique.

Le défi : chaque site web implémente ses formulaires différemment (structures HTML variées, conventions de nommage hétérogènes, contenu dynamique JavaScript, pratiques de sécurité diverses). Concevoir un système universel, robuste et explicable est particulièrement exigeant.

La solution : une API REST basée sur FastAPI qui analyse automatiquement une page web et prépare le remplissage d'un formulaire utilisateur en cinq étapes : Récupération de l'HTML, détection de formulaires, extraction des champs exploitables, mapping avec les champs utilisateur puis pré-remplissage du formulaire.

Le projet respecte trois contraintes fondamentales : aucune interaction avec l'utilisateur final, approche déterministe et interprétable pour garantir la transparence, et architecture suivant les bonnes pratiques (séparation logique métier/API, lisibilité, testabilité, maintenabilité).

2 Architecture et choix techniques

Architecture en trois couches

Le projet suit une architecture orientée services garantissant une séparation claire des responsabilités :

— `routers/` — endpoints HTTP et gestion des entrées/sorties

— `services/` — logique métier complète (scraping, détection, analyse, mapping, remplissage)

— `models/` — schémas de validation Pydantic

Cette organisation rend la logique métier indépendante du framework FastAPI, assurant lisibilité, testabilité et maintenabilité.

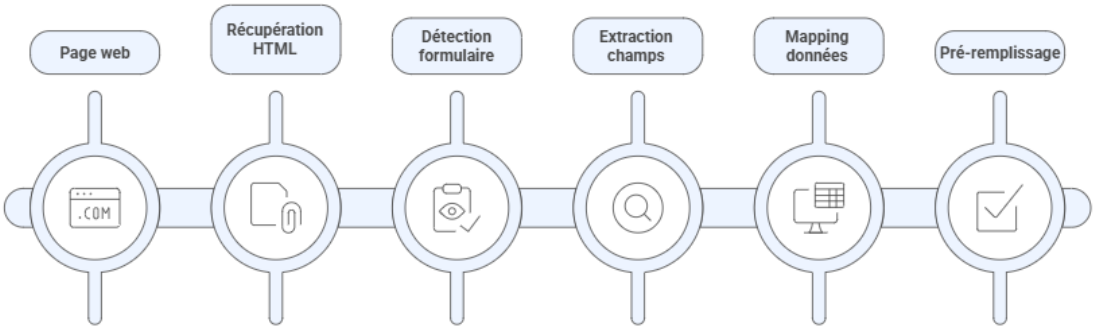
Stack technique

Python 3.11 avec Poetry pour la gestion des dépendances. FastAPI pour l'API (performances, documentation automatique, intégration Pydantic). `requests` et `BeautifulSoup` pour les pages statiques, Selenium en repli pour les cas complexes (contenu dynamique, iframes). Ruff et MyPy garantissent la qualité du code.

Pipeline de traitement

Le système opère en cinq étapes séquentielles : récupération du HTML (HTTP ou Sele-

nium), détection du formulaire (analyse syntaxique + heuristiques), extraction et filtrage des champs, mapping avec les données utilisateur, et pré-remplissage sans soumission.



3 Récupération du HTML et détection du formulaire

Récupération du HTML

La première étape du pipeline consiste à récupérer le code HTML de la page web. Cette tâche est assurée par le module `scraper.py`, dont l'objectif est d'obtenir le contenu de la page quelle que soit la technologie utilisée.

Lorsque cela est possible, une requête HTTP classique est utilisée. Elle permet de récupérer rapidement le code source des pages statiques ou peu dynamiques, tout en simulant différents navigateurs afin de limiter les blocages simples.

Si le formulaire n'apparaît pas dans le code source initial, notamment lorsque le contenu est généré dynamiquement en JavaScript, le système bascule automatiquement vers Selenium. La page est alors chargée dans un navigateur contrôlé, ce qui permet de récupérer le DOM final tel qu'il est visible par l'utilisateur.

Enfin, certains formulaires étant intégrés dans des *iframes*, celles-ci sont analysées lorsque nécessaire. Chaque iframe est explorée indépendamment afin de détecter un formulaire absent du DOM principal.

Détection de la présence d'un formulaire

Une fois le HTML récupéré, il est analysé afin de déterminer si la page contient un formulaire exploitable. La détection repose sur une approche progressive combinant règles simples et heuristiques explicables.

La première étape consiste à rechercher la présence de balises `<form>`, correspondant au cas standard. Le nombre de formulaires détectés est également pris en compte pour identifier les pages contenant plusieurs formulaires.

En complément, la présence de champs de saisie (`input`, `select`, `textarea`) est analysée afin d'identifier des formulaires implémentés sans balise `<form>` explicite.

Pour couvrir les implémentations non conventionnelles, des heuristiques supplémentaires sont appliquées, comme la détection de boutons d'action (ex. : envoyer, s'inscrire), de *placeholders* ou de liens explicites entre labels et champs de saisie.

L'ensemble de ces éléments est ensuite agrégé pour produire un diagnostic structuré indiquant si un formulaire est présent, probable, et sur quels critères repose cette décision. Cette approche garantit une détection robuste tout en restant interprétable.

Endpoint `/form/detect`

Entrée

— `url` : URL de la page web à analyser. Le HTML de cette page est automatiquement récupéré avant l'analyse.

Sortie

— `url` : URL analysée.

— `http_status` : code de statut HTTP retourné lors de la récupération de la page.

— `has_form` : indique si une balise `<form>` est présente.

— `forms_count` : nombre de formulaires détectés dans la page.

— `has_inputs` : indique la présence de champs de saisie (`input`, `select`, `textarea`).

— `probable_form` : indique si une structure de type formulaire est détectée par heuristiques. (True si la page contient au moins un champ de saisie (input, select ou textarea) et au moins un indice fort de formulaire, tel qu'un bouton d'action) ou un élément sémantique (placeholder ou label).

— `reasons` : liste textuelle justifiant la décision de détection.

Perspectives d'évolution

Les évolutions envisagées incluent l'enrichissement du dictionnaire d'heuristiques à partir de nouveaux patterns observés, l'intégration de modèles de vision pour détecter visuellement les formulaires sur des captures d'écran, ainsi qu'une gestion plus fine des interactions modales complexes.

4 Analyse et extraction des champs

Une fois la présence d'un formulaire établie, le HTML est analysé afin d'identifier les champs réellement destinés à une saisie utilisateur. L'extraction repose sur une approche progressive combinant collecte exhaustive et filtrage explicite.

Dans un premier temps, tous les éléments de saisie (`input`, `select`, `textarea`) sont collectés dans le DOM. Chaque élément est ensuite évalué individuellement afin de déterminer

s'il correspond à un champ remplissable par un utilisateur.

Pour chaque champ potentiel, une représentation unifiée `FormField` est construite. Elle regroupe les informations essentielles à l'analyse : balise HTML, type, attributs `name` et `id`, *placeholder* et label associé. Cette modélisation homogène permet de standardiser les traitements ultérieurs.

L'extraction du label, indispensable à l'interprétation sémantique du champ, repose sur une série d'heuristiques simples appliquées séquentiellement : association via l'attribut `for` d'une balise `<label>`, imbrication directe dans un `<label>`, utilisation du *placeholder*, puis récupération du texte proche dans la structure HTML. Cette stratégie permet de s'adapter à la diversité des implémentations rencontrées.

Filtrage des champs non pertinents

L'extraction brute du DOM inclut de nombreux champs techniques qui ne doivent pas être remplis automatiquement, tels que des champs cachés, des jetons de sécurité (*CSRF*), des mécanismes de vérification (*captcha*) ou des dispositifs de suivi (*tracking*, *analytics*).

Un filtrage strict est donc appliqué afin de ne conserver que les champs textuels pertinents. Seuls les types courants (`text`, `email`, `number`, `tel`, `date`) sont autorisés, tandis que les types non exploitables (`hidden`, `password`, `submit`, `button`, `file`, etc.) sont systématiquement exclus.

En complément, la présence de mots-clés caractéristiques de champs techniques dans les attributs `name` ou `id` entraîne l'exclusion automatique du champ. Enfin, un champ n'est conservé que s'il dispose d'au moins un indice sémantique exploitable (`name`, label ou *placeholder*), garantissant une interprétation fiable.

Endpoint `/form/analyze`

Entrée

— `url` : URL de la page web à analyser. Le HTML est automatiquement récupéré avant l'analyse.

Sortie

— `url` : URL analysée.

— `fields_count` : nombre de champs utilisateur identifiés.

— `fields` : liste des champs extraits, chacun décrit par sa structure `FormField`.

Limites et perspectives

L'analyse est volontairement limitée aux champs textuels afin de garantir un comportement stable et prévisible. Les champs de type `checkbox` ou `radio`, ainsi que les champs

multi-composants (dates fragmentées), nécessiteraient une logique d'interprétation plus complexe et ne sont pas couverts à ce stade.

Des évolutions futures pourraient inclure une prise en charge plus fine de ces structures, ainsi qu'un enrichissement progressif des heuristiques à partir de nouveaux formulaires observés.

5 Mapping des champs et données utilisateur

Gestion des données utilisateur

L'API met à disposition plusieurs endpoints REST permettant de gérer les données utilisateur nécessaires au remplissage automatique des formulaires (`POST`, `GET`, `PUT`, `PATCH`, `DELETE`).

Par choix pédagogique et pour simplifier l'architecture, les données sont stockées en mémoire. Un seul utilisateur est géré à la fois et les données sont réinitialisées à chaque redémarrage de l'application. Ce compromis évite l'usage d'une base de données persistante tout en facilitant les tests via l'interface Swagger, avec des valeurs rapidement modifiables.

Principe général du mapping

Une fois les champs du formulaire extraits, l'objectif est d'associer chaque champ à une donnée utilisateur pertinente (email, prénom, téléphone, adresse, etc.).

Le mapping repose sur une approche heuristique déterministe et explicable. Chaque champ est analysé indépendamment et comparé aux clés du modèle `UserData`. Cette comparaison s'appuie sur les informations disponibles dans le champ : `type`, `name`, `id`, *placeholder* et label.

Logique de correspondance (vue simplifiée)

Pour chaque champ extrait, la logique suivante est appliquée :

Champ du formulaire → analyse du `type` → analyse du texte (`name`, `id`, label, *placeholder*) → recherche de synonymes → association avec une clé utilisateur + score de confiance

Étape 1 : priorité au type du champ

Certains types HTML fournissent une information très fiable et sont traités en priorité.

— `type="email"` → `email` (confiance élevée)

— `type="tel"` → `phone`

— `type="date"` → `birth_date`

Exemple :

```
<input type="email" name="contact">
→ email (confiance = 1.0)
```

Étape 2 : analyse sémantique par mots-clés

Si le type n'est pas suffisant ou ambigu, le texte du champ est analysé. Les attributs `name`, `id`, label et *placeholder* sont normalisés, puis comparés à un dictionnaire de synonymes multilingue.

Exemples :

```
<input type="text" name="prenom">
→ first_name

<input type="text" placeholder="Code postal">
→ postal_code
```

Cette approche permet de couvrir de nombreuses variations de nommage sans dépendre d'une convention HTML stricte.

Score de confiance

Chaque correspondance est accompagnée d'un score de confiance compris entre 0 et 1. Ce score reflète la fiabilité de l'association effectuée.

— **1.0** : correspondance certaine (ex. `type="email"`)
— **0.9 – 0.95** : correspondance forte basée sur des mots-clés explicites
— **0.0** : aucune correspondance fiable détectée

Exemples :

```
<input type="email">
→ email | confiance = 1.0 | Matched by input type=email

<input type="text" name="mobile_phone">
→ phone | confiance = 0.9 | Matched by token 'phone'
```

Le score permet de rendre la décision interprétable et de mesurer le degré de certitude associé à chaque mapping.

Gestion des ambiguïtés

Certains champs restent naturellement ambigus, notamment les champs numériques (`type="number"`), qui peuvent correspondre à un âge, un code postal ou un numéro de rue. Dans ces cas, la décision repose uniquement sur les indices textuels disponibles.

Si aucune correspondance fiable n'est trouvée, le champ est explicitement marqué comme non associé.

Endpoint `/form/map`

Entrée

— `url` : URL de la page contenant le formulaire à analyser.

— `user_data` : ensemble des données utilisateur disponibles (identité, contact, adresse, etc.), utilisé comme référence pour le mapping.

Sortie

— `url` : URL analysée.

— `total_fields` : nombre total de champs utilisateur détectés dans le formulaire.

— `matched_fields` : nombre de champs pour lesquels une correspondance avec une donnée utilisateur a été établie.

— `fields` : liste détaillée des champs analysés. Chaque champ est enrichi avec :

  — `matched_key` : clé du modèle `UserData` associée au champ, si une correspondance est trouvée.

  — `confidence` : score de confiance compris entre 0 et 1 indiquant la fiabilité de l'association.

  — `reason` : justification textuelle expliquant la règle ayant conduit au mapping.

Choix techniques et pédagogiques

Le recours à des modèles de langage ou à des approches probabilistes complexes a été volontairement écarté. Le mapping repose exclusivement sur des règles explicites, justifiées et reproductibles.

Chaque association produite par l'API est accompagnée :

— d'une clé utilisateur correspondante,

— d'un score de confiance,

— d'une justification textuelle.

Ce choix garantit une compréhension claire du comportement du système, tant pour l'utilisateur que pour l'évaluation académique.

6 Remplissage automatique et limites techniques

Principe du remplissage automatique

Le remplissage automatique repose sur l'utilisation de Selenium en mode *headless*, permettant de simuler un navigateur réel et d'interagir directement avec le DOM après rendu complet de la page.

Une fois la page chargée, les éléments de saisie (`input`, `select`, `textarea`) sont parcourus dans l'ordre du DOM. Chaque champ est reconstruit sous forme de `FormField`, puis associé à une donnée utilisateur à l'aide des mêmes heuristiques de mapping que celles décrites précédemment.

Lorsqu'une correspondance est trouvée et qu'une valeur utilisateur est disponible, celle-ci est injectée dans le champ à l'aide des méthodes Selenium appropriées. Les champs `select` font l'objet d'un traitement spécifique afin de sélectionner l'option correspondant à la valeur fournie. Aucune soumission du formulaire n'est déclenchée.

Gestion des obstacles courants

De nombreux sites affichent des bannières de consentement aux cookies bloquant toute interaction. Une étape préalable tente donc d'identifier et de valider automatiquement ces bannières en cliquant sur des boutons de consentement courants (ex. : accepter, agree, ok), sans modifier directement le DOM.

Cette approche garantit un comportement proche de celui d'un utilisateur réel et limite les blocages lors du remplissage.

Résultat du remplissage

Pour chaque champ analysé, le système retourne une représentation structurée indiquant :
— la donnée utilisateur associée, si une correspondance a été trouvée,
— le score de confiance du mapping,
— la règle ayant conduit à l'association,
— le statut du remplissage (`filled` : succès ou échec).

Cette granularité permet de distinguer clairement les champs non mappés, les champs mappés mais non remplis, et les champs correctement complétés.

Limite structurelle rencontrée

Une contrainte fondamentale empêche le renvoi d'une page web interactive via une API REST stateless. Selenium opère dans un environnement de navigation isolé côté serveur,

distinct du navigateur de l'utilisateur appelant l'API.

Ainsi, bien que le formulaire soit effectivement rempli dans le navigateur contrôlé par Selenium, cette session ne peut être transmise ou partagée avec le client. L'utilisateur ne peut donc pas poursuivre l'interaction à partir de l'API seule.

Solution retenue et justification

Afin de respecter les contraintes d'une API REST stateless, l'API se limite à retourner une représentation structurée du résultat du remplissage, incluant le nombre de champs détectés, le nombre de champs effectivement remplis et le détail champ par champ. Le remplissage visuel est illustré séparément via un notebook de démonstration, exécuté dans un contexte interactif.

Cette séparation nette entre automatisation backend et interaction utilisateur garantit une architecture cohérente, maintenable et conforme aux principes REST. L'API se concentre exclusivement sur l'analyse, le mapping et la préparation du remplissage, évitant ainsi des solutions hybrides fragiles et assurant un cadre pédagogique clair, reproductible et explicable.

7 API, tests et qualité

L'API s'articule autour d'endpoints spécialisés correspondant chacun à une étape du pipeline : détection, analyse, mapping, remplissage et gestion utilisateur. Cette spécialisation facilite la compréhension et permet des tests indépendants par fonctionnalité.

La documentation générée automatiquement via Swagger offre une interface interactive pour tester les endpoints et visualiser les schémas. Un endpoint `/health` vérifie l'état de l'API en contexte de déploiement.

La qualité repose sur une structure claire avec séparation stricte des responsabilités. Ruff (linting), MyPy (typage statique) et Pytest garantissent robustesse et lisibilité.

Le développement s'appuie sur un dépôt GitHub structuré avec branches, commits réguliers et *pull requests*, traçant les contributions et reflétant des pratiques professionnelles collaboratives.

8 Limites et perspectives d'évolution

8.1 Limites actuelles

Les captchas constituent un frein majeur à l'automatisation complète. Les formulaires fortement dynamiques avec interactions complexes ou rendus conditionnels restent difficilement

analysables. Les protections anti-robots peuvent bloquer l'accès, limitant l'efficacité. L'API se concentre sur la préparation sans interaction graphique, excluant l'expérience utilisateur de son périmètre.

8.2 Perspectives d'évolution

Une extension navigateur rapprocherait le système de l'utilisateur final, bien qu'hors périmètre backend. L'intégration d'API de résolution de captchas améliorerait la couverture fonctionnelle, sous réserve de considérations légales et éthiques.

À long terme, les modèles de vision ou multimodaux (VLM) offriraient des perspectives pour la détection visuelle. Un recours ponctuel aux modèles de langage pourrait gérer les cas ambigus. La gestion multi-utilisateurs et la persistance des données constitueraient des extensions naturelles du système actuel.