

Projet Jardin Solidaire

Titre professionnel

Concepteur développeur d'applications Niveau VI



Table des matières

1	Présentation du projet (FR)	5
2	Présentation du projet (EN)	7
3	Listes des compétences du référentiel	9
4	Spécifications fonctionnelles	13
5	Charte graphique	21
6	Wireframes et maquettes	25
7	Base de données	31
8	Spécifications techniques du projet	35
9	Gestion de projet	39
10	Développement	41
11	CI/CD, Tests et Qualité	59
12	Tests unitaires et tests d'intégration	61
13	Tests end-to-end (E2E)	67
14	Déploiement	71
15	Limites, améliorations et perspectives	75
16	Conclusion	77
	Annexe A — Questionnaire complet	79
	Annexe B — Résultats détaillés du questionnaire	88
	Annexe C — Cahier des charges	99

Chapitre 1

Présentation du projet (FR)

Après plusieurs années en restauration, en vente et dans le social, et attirée depuis longtemps par le **développement web**, j'ai choisi en 2023 de me réorienter et de me former à ce métier.

J'ai intégré **Ada Tech School**, une formation en deux temps : 9 mois à l'école, puis 12 mois en alternance, menant au titre RNCP niveau 6 **Concepteur·rice Développeur·se d'Applications**. J'ai réalisé mon alternance chez **Julaya**, une fintech B2B, dans un environnement de production avec une stack moderne (Next.js, TypeScript, PostgreSQL) et une organisation agile.

Cette expérience m'a appris à travailler sur un produit réel : avancer par itérations, intégrer les retours, traiter les cas limites, et livrer du code **maintenable**.

C'est dans ce cadre que j'ai développé **Jardin Solidaire**.

À l'origine, je suis partie d'un constat simple : les espaces verts améliorent le bien-être et le lien social, mais beaucoup de jardins privés restent sous-utilisés. En parallèle, de nombreuses personnes aimeraient jardiner, apprendre, respirer, se reconnecter au vivant, sans forcément avoir accès à un jardin, surtout en ville. J'ai voulu construire une solution qui relie ces deux réalités, avec une approche **concrète et simple**.

Jardin Solidaire est une plateforme web d'entraide qui met en relation des **propriétaires de jardins** et des **jardinier.es volontaires**. L'idée n'est pas de créer un service marchand : il n'y a pas de paiement. Je mise sur le temps partagé, la coopération et la transmission, pour remettre en vie des jardins qui dorment et faciliter des rencontres de proximité.

Le projet a été initié à trois avec des camarades de ma promotion. À mi-parcours, nous avons choisi de nous séparer pour mener chacune une version aboutie, tout en gardant la même intention de départ.

Aujourd'hui, Jardin Solidaire permet de créer un profil propriétaire ou jardinier.e, de renseigner des informations utiles (photos, localisation, besoins ou compétences), de consulter un listing de jardins avec filtres et favoris, d'accéder à des fiches détaillées, et d'organiser une intervention grâce à des créneaux planifiés via un calendrier et une messagerie liée à l'échange.

Jardin Solidaire se situe au croisement de la **tech**, du **social** et de l'**écologie** : une solution simple pour créer du lien, partager des gestes, et prendre soin des espaces verts de quartier.

Chapitre 2

Présentation du projet (EN)

After several years working in restaurants, retail, and the social sector, and having long been drawn to **web development**, I decided in 2023 to change careers and train in this field.

I joined **Ada Tech School**, a two-part program : a 9-month full-time module at school, followed by a 12-month apprenticeship, leading to the RNCP Level 6 qualification **Application Designer and Developer**. I completed my apprenticeship at **Julaya**, a B2B fintech company, working in production with a modern stack (Next.js, TypeScript, PostgreSQL) in an agile environment.

This experience taught me how to build in a real product context : iterate step by step, integrate feedback, handle edge cases, and deliver **maintainable** code.

In this context, I developed **Jardin Solidaire**.

I started from a simple observation : green spaces improve well-being and social connection, yet many private gardens remain underused. At the same time, many people want to garden, learn, breathe, and reconnect with nature, without having access to a garden, especially in cities. I wanted to connect these two realities through a **practical** and **simple** product.

Jardin Solidaire is a mutual-aid web platform that connects **garden owners** with **volunteer gardeners**. The goal is not to build a commercial service : there is no payment. It focuses on shared time, cooperation, and knowledge transmission, bringing sleeping gardens back to life and enabling local connections.

The project initially started as a team of three (me and two classmates). Midway through, we decided to continue separately, and each of us delivered a complete version while keeping the same original intention.

At its current stage, Jardin Solidaire allows users to create an owner or gardener profile, provide useful information (photos, location, needs or skills), browse a garden listing with filters and favorites, access clear detail pages, and organize a visit through time-slot scheduling with a calendar and a messaging system tied to the exchange.

Jardin Solidaire sits at the intersection of **tech**, **social impact**, and **ecology** : a simple way to build connections, share skills, and take care of neighborhood green spaces.

Chapitre 3

Listes des compétences du référentiel

3.1 Bloc 1 — Développer une application sécurisée

3.1.1 Installation et configuration de l'environnement

Pour lancer Jardin Solidaire, j'ai mis en place un environnement complet et **reproductible** : Node.js, PostgreSQL, Docker, Git et VS Code. J'ai séparé le projet en deux applications (front Next.js et back Express/Node) avec des fichiers .env distincts, afin d'isoler les configurations et de garder les **variables sensibles hors du code**. J'ai aussi documenté tout le parcours d'installation dans un README (clone, docker-compose, migrations Prisma, lancement front/back), pour qu'une autre personne puisse démarrer le projet simplement.

3.1.2 Développement d'interfaces utilisateur.ices

J'ai développé l'interface en React/Next.js en pensant dès le départ aux deux profils principaux : **propriétaires** et **jardinier.es**. J'ai construit les écrans nécessaires au **parcours réel** : recherche et consultation de jardins, fiche détaillée, demande de réservation, gestion des disponibilités et des favoris. J'ai fait attention à la **responsivité (mobile-first)** et à une base d'**accessibilité** (labels de formulaires, contrastes, messages d'erreur clairs), pour que l'application reste utilisable sur différents supports et que l'utilisateur.ice comprenne toujours ce qui se passe.

3.1.3 Développement des composants métier

Côté back-end, j'ai structuré la logique métier pour qu'elle soit **centralisée et réutilisable** : les règles sensibles (réservations, conflits d'horaires, statuts, rôles propriétaire/jardinier.es) sont regroupées dans des fonctions dédiées plutôt que dispersées dans les routes. Ce choix me permet de garder des routes lisibles et de sécuriser la logique **au même endroit**. Côté front, j'ai créé des composants réutilisables orientés métier (carte jardin, filtres, formulaire de réservation) afin d'éviter les duplications et de maintenir une interface **cohérente**.

3.1.4 Gestion de projet

J'ai piloté Jardin Solidaire comme un mini-projet professionnel : définition du périmètre **MVP**, découpage en tâches concrètes, et **priorisation** pour livrer d'abord le parcours essentiel. J'ai utilisé Git/GitHub de façon structurée (branches par fonctionnalité, commits réguliers, Pull Requests) afin d'avoir un historique clair, de limiter les retours en arrière et de garder une base de code **propre**. J'ai également gardé une trace de mes choix et arbitrages (notes / journal), pour **justifier les décisions** prises au fil du développement.

3.2 Bloc 2—Concevoir et développer une application sécurisée organisée en couches

3.2.1 Analyse des besoins

J'ai démarré par l'analyse des besoins des deux profils cibles : le·la propriétaire qui souhaite partager son jardin et le·la jardinier.e qui cherche un lieu pour jardiner. J'ai transformé ces besoins en **user stories** et j'ai priorisé avec une approche type **MoSCoW** afin de protéger le MVP : livrer d'abord ce qui rend l'application utile (**découverte et réservation**), puis garder pour plus tard les évolutions non bloquantes (messagerie plus avancée, système de valorisation).

3.2.2 Architecture logicielle

J'ai conçu Jardin Solidaire avec une **architecture en couches**, pour séparer clairement les responsabilités :

- un front-end Next.js/React pour l'interface ;
- un back-end Express/Node exposant une **API REST** ;
- une base de données PostgreSQL manipulée via **Prisma**.

Dans le back-end, j'ai séparé routes, contrôleurs et logique métier pour garder un code **lisible, testable** et facile à faire évoluer. L'authentification par token et la gestion des rôles sont intégrées dans ce découpage, afin que les contrôles critiques soient **systématiques côté serveur**.

3.2.3 Conception de la base de données

Avant d'implémenter, j'ai modélisé les entités et leurs relations (utilisateur.ice, Jardin, Réservation, Disponibilité, Favori, Avis, etc.), puis j'ai traduit cette conception dans Prisma pour PostgreSQL. J'ai fait ce travail pour garantir la **cohérence du modèle** : relations claires, contraintes utiles (unicité de l'e-mail, champs obligatoires), et structure adaptée aux cas d'usage. J'ai également anticipé certains champs nécessaires à la recherche (ville, code postal, type de jardin), afin de **préparer les évolutions** sans casser la base.

3.2.4 Composants d'accès aux données

J'ai utilisé l'ORM Prisma comme **couche d'accès aux données** pour éviter de disperser du SQL dans le code et limiter les erreurs. Les opérations CRUD et les requêtes plus complexes sont regroupées dans des fonctions dédiées (approche *repository*), ce qui **centralise** la logique d'accès, évite la duplication et facilite la maintenance (optimisations, jointures, pagination).

3.2.5 Documentation technique

J'ai documenté l'essentiel pour qu'une autre personne puisse comprendre et reprendre le projet : structure du repository (front/back), rôle des dossiers clés, et principaux endpoints de l'API (paramètres attendus, réponses, codes d'erreur). Mon objectif était de rendre l'intégration **simple** : comprendre rapidement comment appeler l'API et où se situe la logique métier.

3.2.6 Stratégie de déploiement

J'ai pensé le déploiement en séparant les briques : le front Next.js sur une plateforme type Vercel, API Express déployée séparément, et base de données PostgreSQL gérée à part. Les variables d'environnement (URL DB, secret JWT, CORS, etc.) sont conservées **hors du code** dans un fichier .env. Cette stratégie me permet d'avoir un environnement local clair et un environnement de production **plus sûr** (HTTPS, accès restreints, secrets protégés), tout en gardant une architecture de pratiques professionnelles.

3.3 Bloc 3 — Préparer le déploiement d'une application sécurisée

3.3.1 Tests de l'application

Pour sécuriser l'évolution de Jardin Solidaire, j'ai mis en place une stratégie de tests à plusieurs niveaux : **tests unitaires** pour verrouiller la logique critique, **tests d'intégration** pour valider certaines routes API, et **tests end-to-end** pour rejouer les parcours principaux comme un.e utilisateur.ice.

Je m'en sers comme d'un **filet de sécurité** : dès que je touche à une partie sensible (réservation, statuts, validations), je relance la suite, et elle s'exécute aussi dans la **CI**. Ça me permet de repérer une **régession** immédiatement, et de corriger avant qu'un bug n'arrive en production ou ne se propage dans d'autres fonctionnalités.

3.3.2 Documentation et déploiement

J'ai rédigé un README complet qui centralise toute la documentation du projet afin qu'il puisse être repris facilement. J'y décris la structure front/back, les prérequis, les variables d'environnement, les commandes clés (installation, migrations Prisma, lancement en local), ainsi qu'une procédure de déploiement pas à pas. L'objectif est simple : permettre à quelqu'un de cloner le dépôt, lancer l'application et la déployer, sans avoir à deviner les réglages.

3.3.3 Démarche DevOps

J'ai appliqué une démarche inspirée **DevOps** : code centralisé sur GitHub, branches par fonctionnalité, et **intégration continue** via GitHub Actions. Cette automatisation détecte rapidement les problèmes de build, de dépendances ou de configuration. Les secrets (tokens, variables sensibles) restent dans la configuration des plateformes/outils et ne sont **jamais commités** dans le dépôt.

3.4 Éléments transversaux à vérifier

3.4.1 Qualité du code et sécurité

J'ai cherché à garder un code **lisible** et homogène (naming, organisation, indentation) et j'ai utilisé des outils de vérification (linter). Pour la sécurité, j'ai **hashé les mots de passe**, validé les entrées, empêché les injections SQL via l'ORM, et appliqué les bonnes pratiques de base : permissions par rôle, authentification obligatoire pour les actions sensibles, secrets non exposés dans le repository.

3.4.2 Documentation et présentation

Le projet est accompagné d'un dossier structuré (contexte, besoins, choix techniques, architecture) et de supports visuels (diagrammes, captures d'écran, extraits de code). Je m'en sers pour montrer la **cohérence** entre le besoin, la conception et l'implémentation, et pour rendre la soutenance plus concrète.

3.4.3 Compétences en anglais

J'ai produit un résumé en anglais de Jardin Solidaire (objectif, cible, stack, architecture). Une partie de la documentation (README), du nommage et des pratiques Git est également en anglais.

3.4.4 Validation des besoins

Tout au long du développement, j'ai vérifié que les fonctionnalités livrées répondaient bien au **besoin initial** : parcours de demande de réservation compréhensible, gestion des disponibilités côté propriétaire, lisibilité des informations côté jardinier.es. J'ai aussi pris en compte les retours pour ajuster certaines priorités (clarification de formulaires, simplification d'écrans), afin de rester alignée avec un usage réel.

3.4.5 Organisation et architecture

J'ai organisé l'application en couches : interface (React/Next.js), logique métier (services/contrôleurs), et accès aux données (Prisma/PostgreSQL). Côté front, j'ai structuré par pages et par domaines ; côté back, routes → contrôleurs → logique métier, pour faciliter les tests, la lisibilité et les évolutions.

3.4.6 Modèles de données

J'ai produit un modèle conceptuel pour identifier les entités principales et leurs relations, puis je l'ai traduit en schéma technique via Prisma pour générer la base PostgreSQL et suivre l'évolution avec des migrations. Ce lien entre conception et implémentation est **documenté** pour montrer la cohérence du modèle.

3.4.7 Accès aux données

J'ai basé l'accès aux données sur Prisma et une approche de type repository : plutôt que manipuler du SQL en plein code, j'appelle des fonctions de haut niveau (`createBooking`, `findGardensWithFilters`, etc.). Cela **limite les erreurs**, centralise la logique, et facilite les optimisations. Cet accès est protégé au niveau du back-end par les contrôles d'authentification et d'autorisation.

Chapitre 4

Spécifications fonctionnelles

Expression des besoins

Jardin Solidaire vise à résoudre un besoin très concret : aujourd’hui, beaucoup de personnes possèdent un jardin mais n’ont pas toujours le temps, l’énergie ou les ressources pour l’entretenir.

En parallèle, beaucoup d’autres personnes aimeraient jardiner, apprendre, respirer et se reconnecter à la nature, mais n’y ont pas accès, surtout en ville.

L’objectif de l’application est donc de faciliter la **mise en relation** et l’**organisation** entre deux publics : les **propriétaires de jardins** et les **jardinier.es volontaires**.

J’ai choisi une interface web **responsive** pour que le service soit utilisable dans la vraie vie, sur ordinateur comme sur téléphone, sans installation. L’application doit couvrir aussi bien un coup de main ponctuel qu’un échange régulier (suivi saisonnier, rendez-vous récurrents), sans complexifier le parcours.

Dès le cadrage, j’ai pensé à l’**éco-conception** en intégrant une contrainte de **sobriété numérique** : l’application doit rester légère et efficace, notamment sur mobile et avec des connexions variables. L’objectif n’est pas de sur-promettre un impact, mais de limiter les charges inutiles dès la conception (pages rapides à charger, données affichées progressivement, interactions simples et réactives).

Enfin, j’ai fait le choix d’un modèle **non marchand** : pas de paiement, pour rester cohérente avec l’intention du projet. L’échange repose sur le temps partagé, l’entraide et la transmission. L’enjeu principal n’est donc pas d’optimiser un prix, mais de créer un cadre clair, rassurant et efficace pour rendre l’entraide possible.

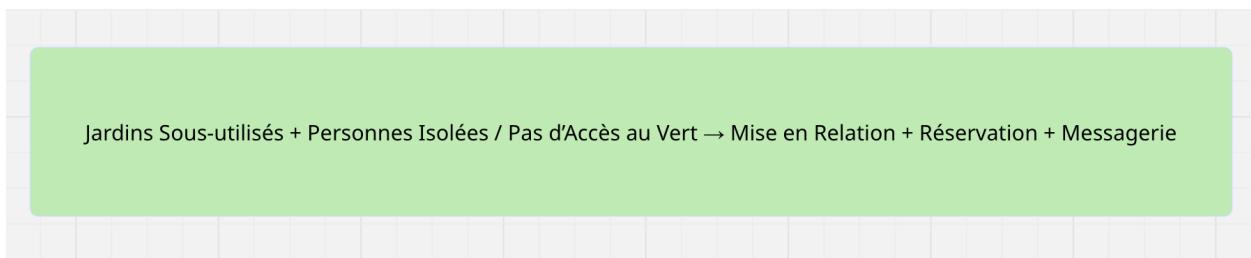


FIGURE 4.1 – Synthèse du besoin auquel répond Jardin Solidaire.

Démarche de cadrage : questionnaire, PESTEL, benchmark

Avant de construire l'application, j'ai cadré le projet pour vérifier qu'il répondait à des attentes réelles. Pour cela, j'ai commencé par un **questionnaire** envoyé autour de moi pour identifier la cible, mais aussi pour comprendre les freins et les attentes des futur.es utilisateur.ices.

J'ai fait attention à l'**accessibilité** des questions : ici, "accessibilité" signifie surtout un langage clair, sans jargon, afin de toucher aussi des personnes moins à l'aise avec le numérique.

Le formulaire comportait une section propriétaire (fréquence d'entretien, disponibilités, types de besoins) et une section jardinier.e (niveau d'expérience, créneaux, motivations, contraintes).

J'ai reçu **27 réponses**, et les conclusions étaient très nettes : les utilisateur.ice.s attendent d'abord de la **simplicité** (fiches claires, consignes précises), ensuite de la **visibilité** (créneaux lisibles sur un calendrier), et enfin de la **confiance** (profils détaillés, messagerie, et à terme avis / notes).

Annexe A — Questionnaire complet

Annexe B — Résultats détaillés du questionnaire

Analyse PESTEL

En complément, j'ai réalisé une **analyse PESTEL**. C'est une méthode qui permet d'analyser un projet dans son contexte global en regardant six dimensions : **Politique, Économique, Sociale, Technologique, Environnementale et Légale**.

L'intérêt est de repérer à la fois les opportunités (par exemple le regain d'intérêt pour le jardinage ou les initiatives d'espaces verts) et les contraintes (par exemple la sécurité des personnes, la protection des données et le cadre légal).

P — Politique Opportunités : <ul style="list-style-type: none">○ Initiatives publiques autour du verdissement et du lien social.○ Possibles partenariats locaux (assos, quartiers). Contraintes : <ul style="list-style-type: none">○ Clarifier le rôle de la plateforme en cas d'incident.○ Définir un cadre en cas de partenariat (responsabilités).	E — Économique Opportunités : <ul style="list-style-type: none">○ Alternative à l'entretien payant.○ Valorise l'échange de temps (entraide). Contraintes : <ul style="list-style-type: none">○ Modèle non marchand : valeur à rendre très claire.○ Besoin d'un parcours simple et efficace (sinon abandon).	S — Social Opportunités : <ul style="list-style-type: none">○ Réduit l'isolement, crée du lien de proximité.○ Accès au vivant pour des personnes sans jardin. Contraintes : <ul style="list-style-type: none">○ Confiance : peur d'accueillir un inconnu.○ Besoin de consignes claires et d'un cadre rassurant.
T — Technologique Opportunités : <ul style="list-style-type: none">○ Web mobile-first : accessible sans installer d'app.○ Carte + calendrier + messagerie : organisation facilitée. Contraintes : <ul style="list-style-type: none">○ Sécurité (auth, données) et fiabilité des réservations.○ Prévention des abus (signalement / modération).	E — Environnemental Opportunités : <ul style="list-style-type: none">○ Jardins "réactivés" : biodiversité locale.○ Sensibilisation à des pratiques plus durables. Contraintes : <ul style="list-style-type: none">○ Ne pas sur-promettre l'impact.○ Définir des indicateurs simples (heures, interventions).	L — Légal Opportunités : <ul style="list-style-type: none">○ RGPD : cadre clair pour protéger les données.○ CGU possibles pour poser des règles simples. Contraintes : <ul style="list-style-type: none">○ Données perso : limiter ce qui est public.○ Responsabilité civile à cadrer (incidents, dégâts).

FIGURE 4.2 – Analyse PESTEL - opportunités et contraintes du projet.

Benchmark

Pour construire Jardin Solidaire sur du concret, j'ai réalisé un **benchmark** : j'ai passé en revue plusieurs plateformes proches (mise en relation, entraide, services entre particuliers). Je me suis concentrée sur trois moments décisifs : **comment on trouve, comment on se met d'accord, et ce qui met en confiance** juste avant de s'engager.

Ce benchmark a guidé mes choix produit, avec comme objectif de réduire les hésitations et donner envie de passer du "je regarde" au "j'y vais".

Plateforme	Cible / promesse	Forces	Limites	Inspirations
PlantezChezNous	Co-jardinage / entraide	Promesse alignée, esprit communauté	Cadre opérationnel et sécurité à expliciter	Promesse simple, règles claires, profils rassurants
AlloVoisins	Services entre voisins	Profils/avis, usage local	Tendance prestation souvent payante	Reprendre les codes de confiance, sans logique marchande
JardinPrivé.fr	Accès à des jardins (location)	Offre visible, accès à des lieux	Modèle payant, peu d'entraide	Se différencier : échange de temps + entraide
Privateaser	Location d'espaces (événements)	Fiches claires, critères lisibles	Pas centré jardinage / entraide	Fiches très lisibles : photos, infos, règles

FIGURE 4.3 – Benchmark - comparaison de solutions proches et enseignements appliqués.

Personas : profils utilisateur.ices

À partir des retours du questionnaire, j'ai construit des **personas** pour garder un repère concret pendant toute la conception. L'idée était simple : ne pas concevoir pour un.e utilisateur.ice abstrait.e, mais pour des personnes réalistes, avec des contraintes et des attentes très différentes.

Un **persona** est un profil fictif, mais basé sur des éléments observés (motivations, freins, habitudes).

Personas



Persona 1 : Louise
Propriétaire de jardin occupée



Persona 2 : Thomas
Le jardinier novice



Persona 3 : Marie et Jean
Les retraités passionnés

FIGURE 4.4 – Personas

Louise, 45 ans représente un profil de propriétaire avec un emploi du temps chargé. Elle veut un jardin entretenu sans devoir organiser toute une logistique. Son frein principal est la confiance : elle a besoin de savoir qui vient chez elle, et souhaite pouvoir consulter des profils clairs et des avis.

Thomas, 28 ans représente un jardinier volontaire qui vit en ville. Il veut jardiner pour apprendre et passer du temps dehors. Il craint de ne pas avoir assez d'expérience et de ne pas trouver de jardin près de chez lui. Il attend donc une recherche simple, idéalement avec une carte et des repères de localisation.

Marie, 67 ans et Jean, 70 ans représentent un couple de propriétaires retraités. Ils ont un grand jardin qu'ils entretenaient depuis des années, mais qui devient difficile à gérer physiquement. Ils souhaitent transmettre leur savoir, mais sont moins à l'aise avec les outils numériques. Leur attente principale est une interface simple, rassurante et intuitive, notamment pour gérer les créneaux et communiquer.

Objectifs fonctionnels majeurs

À partir de ces besoins, j'ai défini les **objectifs fonctionnels majeurs**, c'est-à-dire les grandes actions que l'application doit permettre.

Simplicité d'accès

L'application doit être utilisable depuis un navigateur, sur mobile comme sur ordinateur, et permettre d'accéder rapidement aux actions principales.

Publication des jardins

Un propriétaire doit pouvoir décrire son jardin de façon claire et utile, avec des photos, une localisation approximative, et des informations qui permettent de comprendre ce qui est attendu.

Découverte et mise en relation

Un.e jardinier.e doit pouvoir explorer les jardins, filtrer selon des critères simples, puis envoyer une demande.

Organisation des interventions

L'application doit proposer un calendrier pour afficher et réserver des créneaux, ainsi qu'une messagerie pour préciser les consignes.

Il est aussi important que l'utilisateur.ice comprenne facilement l'état de sa demande : "en attente", "acceptée", "planifiée", "réalisée", "annulée".

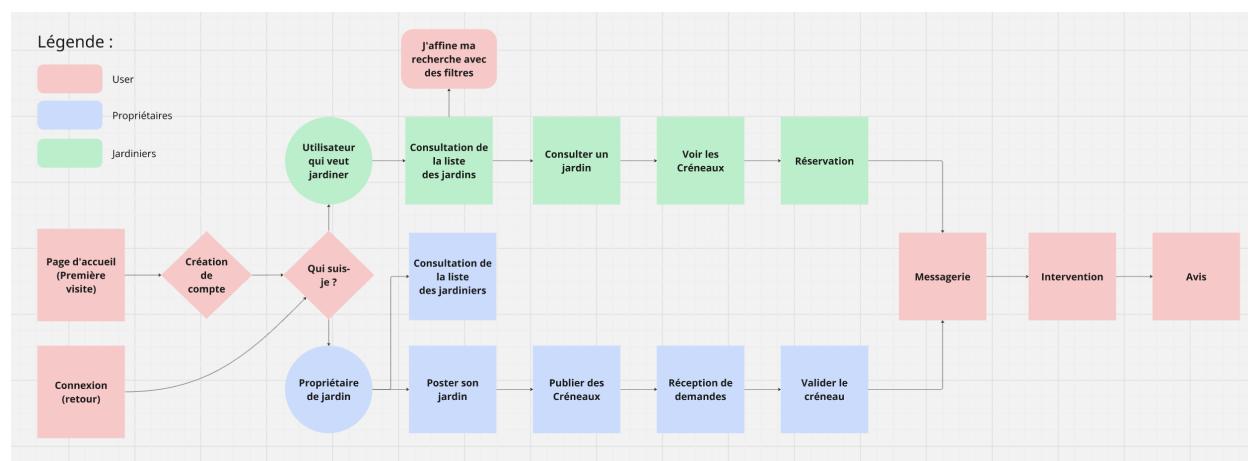


FIGURE 4.5 – Parcours principal - de la découverte à l'organisation de l'intervention.

Fonctionnalités clés (MVP)

À partir des besoins identifiés et des personas, j'ai défini le MVP (*Minimum Viable Product*), c'est-à-dire la **plus petite version** du produit qui permet de **tester l'idée sur le terrain**.

Pour prioriser ce MVP, j'ai utilisé la méthode **MoSCoW**. Cette méthode consiste à classer les fonctionnalités en quatre niveaux de priorité : **Must-have** (indispensables), **Should-have** (importantes mais non indispensables), **Could-have** (optionnelles, à ajouter si le temps le permet), et **Won't-have (for now)** (hors périmètre pour cette version, mais pouvant être envisagées ultérieurement).

Les **Must-have** correspondent au parcours principal complet : créer un compte, publier ou consulter un jardin, choisir un créneau, envoyer une demande, puis suivre son statut. Tout le reste peut venir ensuite.

Les **Should-have** regroupent ce qui améliore fortement l'expérience et la confiance, sans être nécessaire pour valider le cœur du produit : par exemple une recherche plus avancée, une carte, un calendrier plus interactif, des avis, ou une connexion via des services tiers.

Enfin, les **Could-have** correspondent aux ajouts qui enrichissent la communauté une fois que le parcours principal est solide : une banque du temps (cumul d'heures), des badges, ou des récits d'expérience.

L'objectif de cette méthode est donc d'obtenir un produit cohérent, complet et testable dès maintenant, puis de l'enrichir progressivement sans se disperser.

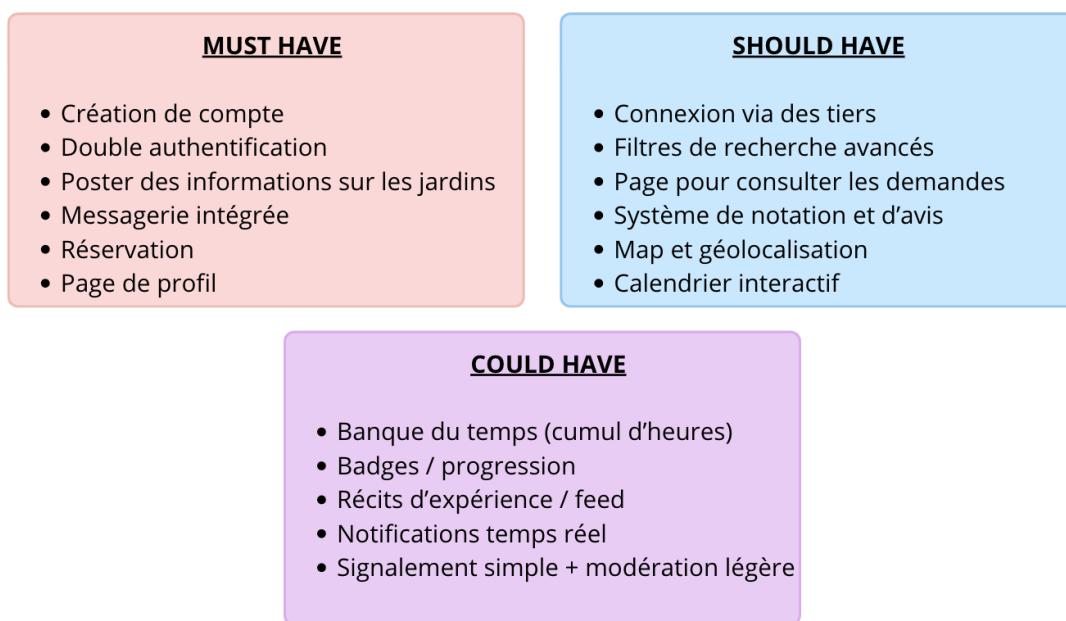


FIGURE 4.6 – Priorisation MoSCoW des fonctionnalités (MVP et évolutions).

Création de compte

Inscription par e-mail et mot de passe, puis validation du compte via un code reçu par e-mail. Les informations du profil (photo, rôle, localisation, besoins ou compétences) sont consultables et modifiables par l'utilisateur.ice.

Connexion

Authentification pour accéder aux fonctionnalités principales (publication, recherche, calendrier, messagerie) et une double authentification (2FA) par code e-mail.

Publication et gestion des jardins (propriétaires)

Création et édition d'une fiche jardin avec description, photos, localisation (approximative), superficie, besoins (tonte, désherbage, arrosage, plantation, etc.) et créneaux disponibles. L'utilisateur.ice contrôle la visibilité de son jardin.

Recherche de jardins (jardinier.es)

Exploration des jardins via une liste et des filtres simples (activité, disponibilités, rayon), avec la possibilité d'ajouter des favoris et d'accéder à des fiches détaillées (photos, consignes, outils disponibles).

Carte

Affichage des jardins sur une carte, en complément de la liste, pour faciliter la recherche à proximité.

Demandes d'aide et réservations

Envoi d'une demande sur un créneau libre. Suivi d'un statut clair : *en attente, acceptée, refusée, planifiée, réalisée* ou *annulée*. Le propriétaire peut accepter ou refuser, et le.la jardinier.e.e peut annuler avant la réalisation.

Messagerie intégrée

Discussion liée à un jardin ou à une réservation, afin de clarifier les consignes (accès, outils, durée, point de rencontre). Des notifications par e-mail sont envoyées en cas de nouveau message.

Tableau de bord et historique

Un espace de suivi permet de retrouver les demandes envoyées (côté jardinier.es) ou reçues (côté propriétaire), ainsi que l'historique des interventions à venir et passées.

Extensions prévues (post-MVP)

Une fois le MVP stable, plusieurs extensions sont envisagées : connexion via Google, système complet d'avis et de notation, notifications en temps réel, cumul d'heures (banque du temps), badges et éléments de gamification.

Chapitre 5

Charte graphique

Intention

Dès le début de Jardin Solidaire, j'ai construit l'identité visuelle avec une idée simple : **faire gagner du temps et installer la confiance**. Je veux qu'en quelques secondes, l'utilisateur.ice comprenne où iel est, ce qu'iel peut faire, et comment avancer.

Le projet s'adresse à des profils très différents : des propriétaires parfois pressé·es, des personnes moins à l'aise avec le numérique, et des jardinier.es volontaires qui utilisent souvent leur téléphone. J'ai donc choisi une charte qui **guide** plutôt qu'elle ne décore : une interface lisible, rassurante et agréable, qui évoque la nature sans surcharge.

Je vise une sensation de calme et de stabilité : des écrans aérés, une hiérarchie nette, et des actions immédiatement repérables. L'objectif est de **réduire les erreurs** et de **fluidifier le parcours**.

Principes

Accessibilité

Le premier pilier est l'**accessibilité**. Je veux que l'application reste utilisable par le plus grand nombre, y compris sur petit écran, en extérieur, ou avec une vision diminuée. Pour ça, j'ai travaillé en priorité la lisibilité : contrastes suffisants, tailles de texte confortables, et éléments interactifs faciles à cibler.

Je me suis appuyée sur les recommandations **WCAG** (AA/AAA) et j'ai vérifié mes contrastes avec *Contrast Checker* afin de m'assurer que les informations essentielles restent lisibles dans des conditions réelles (Figures 5.1). J'ai aussi veillé aux zones cliquables : sur mobile, une action doit être simple à toucher sans précision chirurgicale, sinon on multiplie les erreurs.

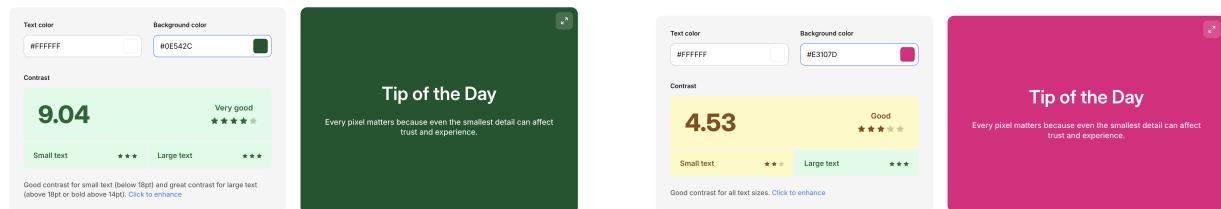


FIGURE 5.1 – Vérifications de contraste : fond vert (lisibilité élevée) et accent fuchsia (CTA visible et lisible).

Le deuxième principe, c'est le **mobile-first**. J'ai conçu les écrans d'abord pour le téléphone, parce que c'est l'usage le plus contraint : peu d'espace, navigation au pouce, attention fragmentée. Mon objectif est de garder une interface **claire** et priorisée : peu d'informations à la fois, mais les bonnes informations, au bon moment.

Je prends en compte la logique de *thumb-reach* : les actions fréquentes doivent rester faciles d'accès, sinon l'expérience devient pénible. Une fois cette base solide, j'enrichis la version desktop avec plus d'espace (colonnes, panneaux), sans changer le langage visuel.

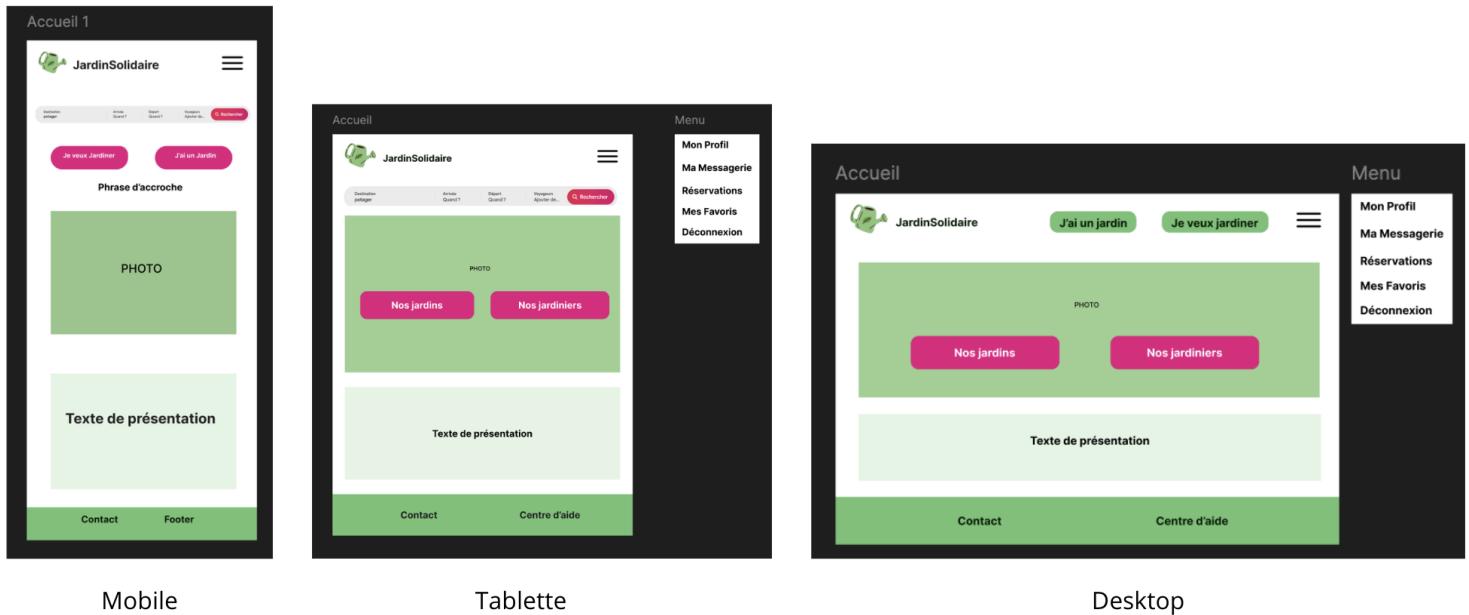


FIGURE 5.2 – Mobile-first : une base mobile enrichie sur grand écran, sans casser les repères.

Cohérence

Le troisième principe, c'est la **cohérence**. Une interface cohérente réduit la charge mentale : l'utilisateur.ice reconnaît les patterns et n'a pas besoin de réapprendre à chaque écran.

Pour y arriver, j'ai construit des composants UI réutilisables (cartes, listes, filtres, bandeaux d'action) avec des espacements réguliers et une iconographie discrète. Mon objectif est que l'application reste stable visuellement : mêmes codes, mêmes repères, mêmes réflexes.

Centralisation dans Figma

Toute la charte est centralisée dans **Figma** : styles de couleurs, styles de texte, et composants. Je m'en sers comme d'une référence unique entre design et développement : quand j'ajoute un écran, je réutilise la même base, avec les mêmes règles.

Ça facilite le *hand-off* et ça évite les dérives au fil du temps : l'interface reste cohérente, même quand le projet grandit.

Éléments de la charte : couleurs, typographie, composants



FIGURE 5.3 – Palettes Jardin Solidaire : couleurs d'identité (verts), accent (fuchsia) et fonds (menthe/blanc).

J'ai volontairement choisi une palette **courte et mémorisable** (Figure 5.3). Le **vert médium** porte l'identité et structure les éléments clés : il évoque la nature, mais surtout il donne des repères stables dans l'interface.

Le **rose fuchsia** sert d'accent : je l'utilise pour mettre en évidence les actions principales et guider l'œil. Quand une personne hésite, je veux qu'elle repère immédiatement où agir.

Pour éviter la fatigue visuelle, j'ajoute un menthe très clair en fond : il aère les écrans et crée des zones de repos. Un vert foncé renforce la lisibilité sur les titres et certains textes, et le blanc reste la base pour conserver une interface lumineuse et simple.

J'ai aussi défini une hiérarchie typographique cohérente (titres, sous-titres, texte, labels). L'idée est de guider la lecture sans obliger l'utilisateur.ice à tout parcourir : on comprend d'abord l'essentiel, puis on détaille si besoin.

Chapitre 6

Wireframes et maquettes

Objectifs

Avant d'écrire du code, j'ai conçu les écrans sur **Figma** pour une raison simple : je voulais que Jardin Solidaire ne soit pas "un assemblage de pages", mais un **parcours fluide** qui transforme une intention ("j'aimerais jardiner") en action concrète ("j'ai une réservation, un suivi, et un cadre d'échange").

J'ai travaillé en deux temps : d'abord des **wireframes** (quoi afficher, dans quel ordre), puis des **maquettes** pour fixer la hiérarchie visuelle, la lisibilité mobile et les états critiques (chargement, erreur, connecté-e/non connecté·e, confirmation). Ces maquettes m'ont servi de **contrat d'implémentation** pendant le développement.

Fil conducteur : du premier écran à la réservation

J'ai organisé les maquettes en suivant le parcours le plus important du produit : **découvrir → comprendre → se connecter → choisir → réserver → suivre et échanger**.

Clarifier le concept dès l'arrivée

L'accueil pose immédiatement le contexte : Jardin Solidaire s'adresse à deux profils (propriétaires et jardinier.es). J'ai donc choisi un onboarding très direct, avec deux entrées claires ("Je veux jardiner" / "Je propose mon jardin"), pour que l'utilisateur.ice sache où aller en quelques secondes (Figure 6.1).



FIGURE 6.1 – Accueil

Réduire la friction sans sacrifier la sécurité

Ensuite, j'ai conçu l'inscription et la connexion avec une contrainte forte : **aller vite**, tout en gardant un accès sécurisé. Les formulaires sont simples et guidés (Figure 6.2 et Figure 6.3). J'ai choisi une **vérification par code e-mail** plutôt qu'un lien : sur mobile, c'est plus lisible, plus immédiat, et ça évite de perdre l'utilisateur.ice dans un changement d'application (mail → navigateur). (Figure 6.4).



FIGURE 6.2 – Création de compte



FIGURE 6.3 – Connexion



FIGURE 6.4 – Vérification par code

Découvrir facilement : liste claire + action évidente

Une fois dans l'application, le besoin principal est la découverte : trouver un jardin facilement. J'ai donc conçu une liste de jardins lisible, avec une structure carte, et la place nécessaire pour des filtres/favoris (Figure 6.5). L'idée est que la personne puisse **scanner** rapidement, retenir, puis revenir.

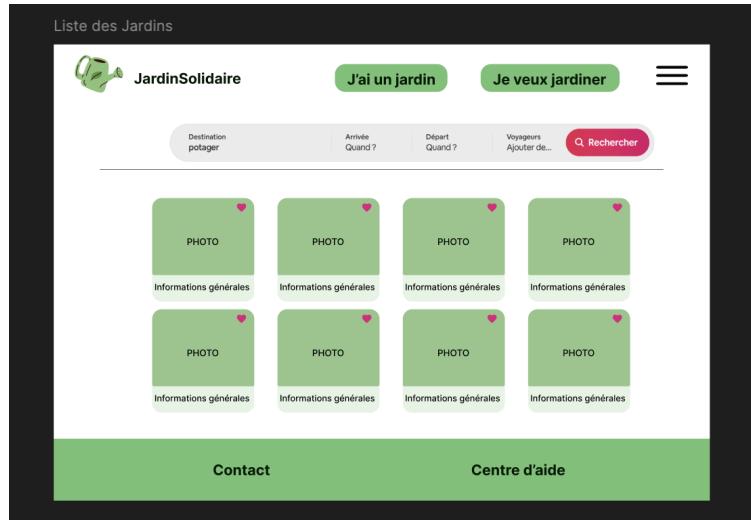


FIGURE 6.5 – Liste des jardins

Rassurer avant d'engager : fiche jardin = contexte + décision

La fiche jardin est l'écran qui transforme l'intérêt en décision. Je l'ai conçue comme un "résumé utile", et surtout **une action principale claire** pour passer à la réservation. L'utilisateur.ice doit comprendre **où iel va, pourquoi iel vient, et dans quelles conditions**, avant d'envoyer une demande (Figure 6.6).

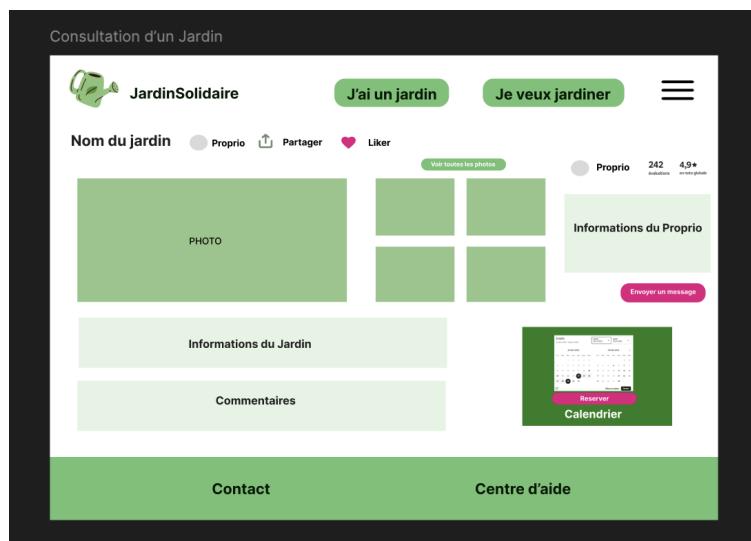


FIGURE 6.6 – Fiche jardin

Le moment critique : réserver

La réservation est la partie la plus sensible du parcours : si l'utilisateur.ice ne comprend pas ce qu'iel réserve, ou s'iel doute, iel abandonne. J'ai donc conçu une page où tout est explicite : la demande, le créneau, et le contexte (jardin/jardinier.e). L'objectif est qu'au moment de cliquer, la personne sache exactement ce qui part côté serveur (Figure 6.7).

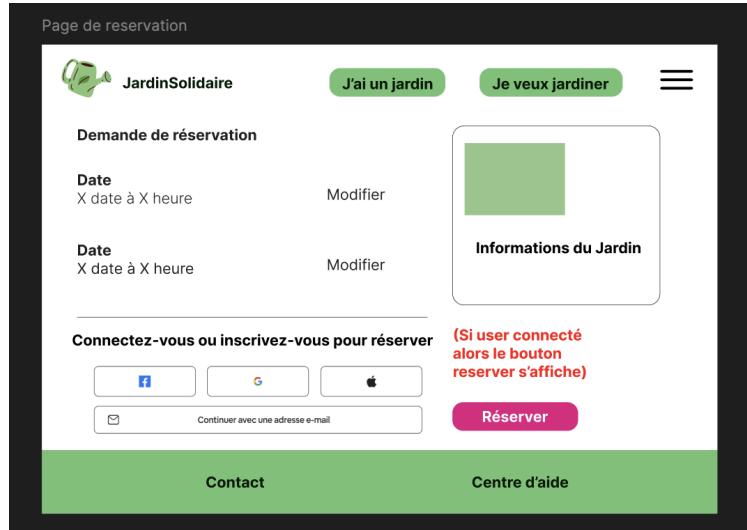


FIGURE 6.7 – Page de réservation

Après l'action : suivre

Une fois la demande envoyée, j'ai prévu des écrans de suivi : une liste de réservations (futures / passées) et un détail de réservation, pour que l'utilisateur.ice puisse vérifier l'état, revenir sur les infos, et agir sans se perdre (Figure 6.8).

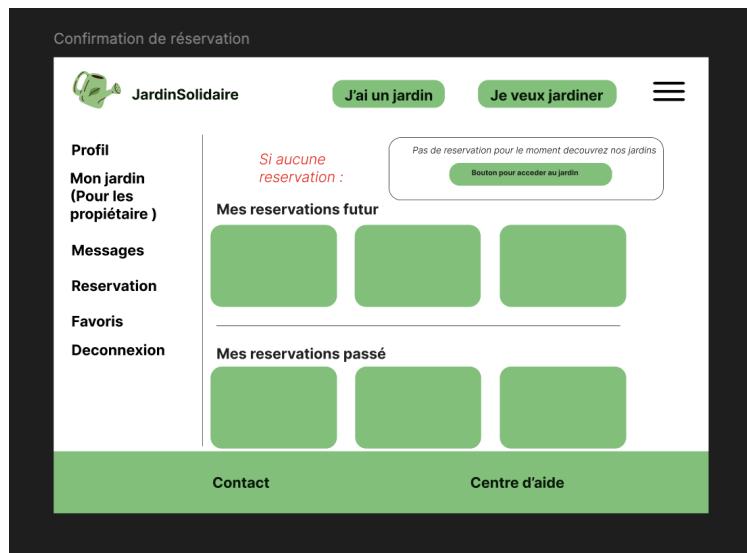


FIGURE 6.8 – Suivi

Échanger sans sortir du cadre : messagerie liée à la réservation

Enfin, j'ai conçu une messagerie qui reste **attachée au contexte** : on discute à propos d'un jardin, d'un créneau, et d'une demande. C'est un choix de fiabilité : moins de malentendus, une trace, et une coordination plus simple (Figure 6.9).



FIGURE 6.9 – Boîte de réception

Écrans "réassurance"

Pour renforcer la confiance, j'ai aussi prévu des écrans qui rassurent et fidélisent : le profil (informations utiles, cohérence), et selon la priorité produit soit les favoris (revenir plus tard), soit l'avis (boucle de confiance).

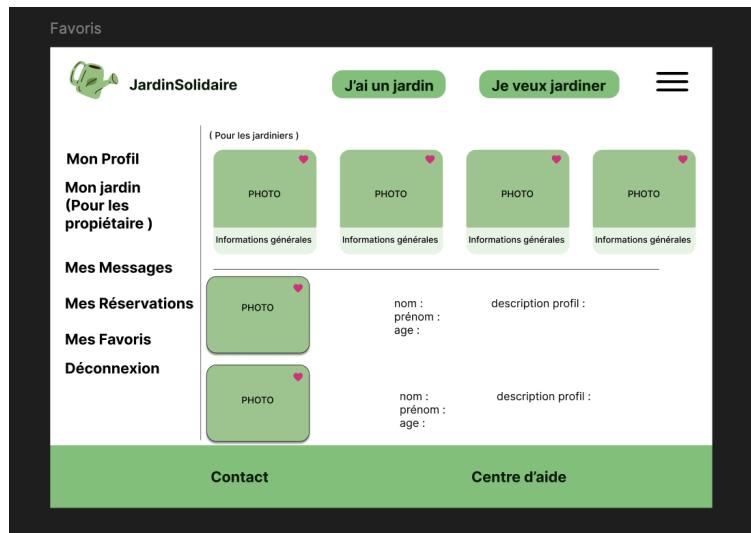


FIGURE 6.10 – Favoris

Chapitre 7

Base de données

De l'idée "métier" à une base réelle : MCD puis MLD

Avant d'implémenter la base de données, j'ai commencé par modéliser les informations de Jardin Solidaire. L'objectif est de ne pas créer des tables au hasard, mais de construire une base qui reflète les besoins du projet : créer un compte, publier un jardin, proposer des créneaux, réserver, échanger, puis éventuellement laisser un avis.

Pour cela, j'ai d'abord réalisé un **MCD** (**Modèle Conceptuel de Données**). Le MCD est une vue "métier" : il décrit ce que l'application doit manipuler, sans entrer dans les détails techniques. On y liste les grandes **entités** du projet et leurs liens. Une entité correspond à un type d'objet important dans l'application. Dans Jardin Solidaire, on retrouve notamment : **utilisateur.ice**, **jardin**, **créneau de disponibilité**, **réservation**, **messagerie** et **avis**. Selon les besoins, on peut aussi ajouter des entités comme les **favoris** ou des tables de liaison.

Sur le MCD (Figure 7.1), chaque bloc représente une entité et chaque lien représente une **relation**. C'est à cette étape que l'on clarifie des règles simples mais essentielles. Par exemple : un propriétaire peut posséder plusieurs jardins, mais un jardin n'a qu'un seul propriétaire. Un jardin peut proposer plusieurs créneaux de disponibilité, et une réservation relie un.e jardinier.e à un jardin sur un créneau précis. Les avis, eux, doivent rester rattachés à une interaction réelle pour conserver un système de confiance cohérent.

Le MCD est donc un outil de **vérification** : il permet de vérifier que le modèle correspond bien au parcours utilisateur.ice et de repérer tôt les incohérences. Par exemple, sans entité "créneau", il est difficile de construire un calendrier clair ; et si l'on mélange "créneau" et "réservation" dans une seule table, la logique devient plus confuse et le risque de conflits augmente.

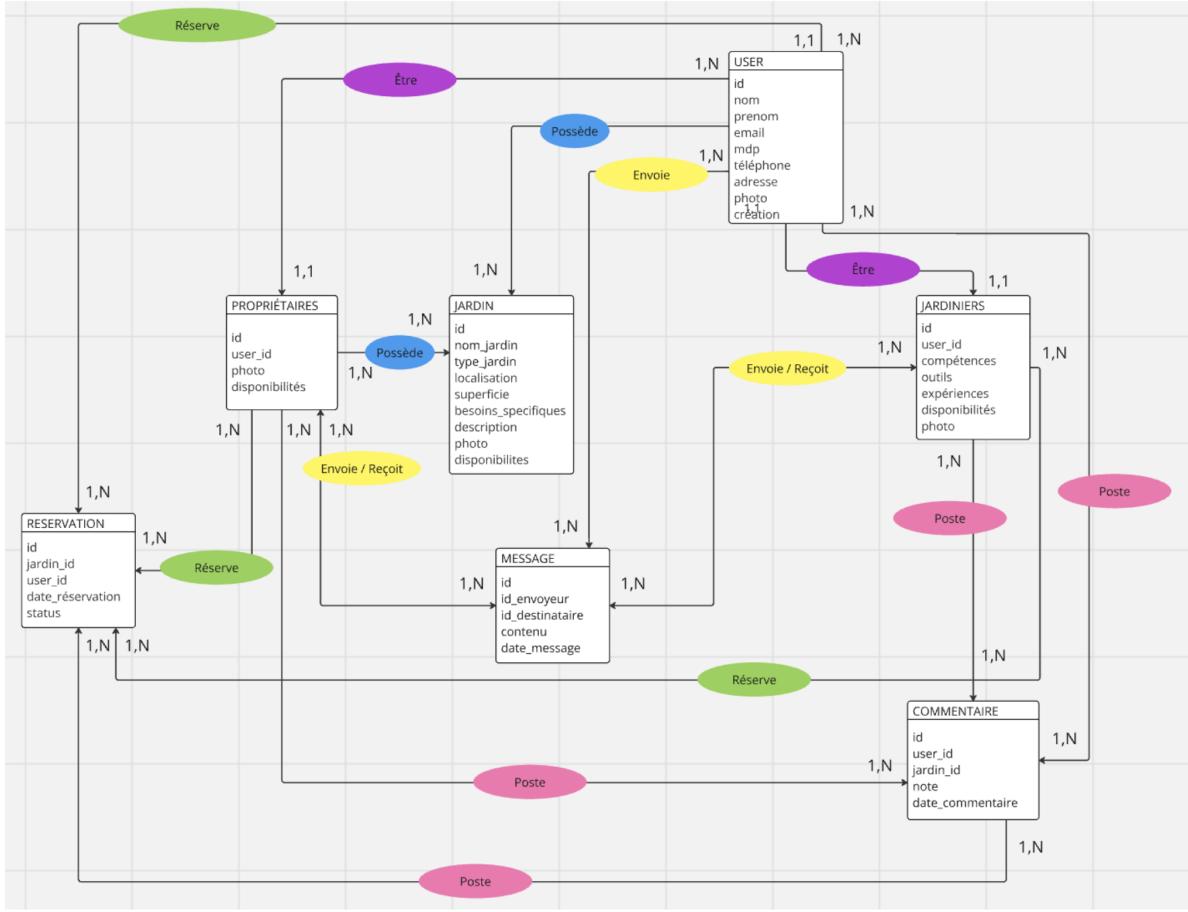


FIGURE 7.1 – MCD — vue métier des entités et de leurs relations.

Une fois le MCD validé, j'ai dérivé un **MLD (Modèle Logique de Données)**. Le MLD est plus proche de l'implémentation : on passe du concept au concret. Chaque entité devient une table, et on précise comment la base va garantir l'intégrité des données.

Dans un MLD, on définit notamment :

- les **clés primaires**, c'est-à-dire l'identifiant unique d'une ligne de table. Par exemple, une réservation a un identifiant unique qui permet de la retrouver sans ambiguïté;
- les **clés étrangères**, c'est-à-dire des champs qui pointent vers une autre table pour créer un lien. Par exemple, une réservation contient l'identifiant de l'utilisateur.ice et l'identifiant du jardin (et/ou du créneau) pour savoir qui a réservé quoi;
- les **contraintes**, qui sont des règles imposées par la base. Une contrainte classique est l'unicité de l'**e-mail** : cela évite d'avoir deux comptes avec la même adresse. On peut aussi imposer que certains champs soient obligatoires (par exemple un jardin doit avoir un titre), ou que certains statuts appartiennent à une liste précise.

On retrouve aussi la notion de **cardinalité**, qui décrit le nombre possible de relations. Par exemple, "un propriétaire → plusieurs jardins" est une cardinalité de type **1-n**. Cela aide à construire la base correctement et à éviter des relations impossibles ou ambiguës.

Le MLD joue alors le rôle de **contrat** : c'est ce schéma (Figure 7.2) qui m'a guidée pour créer une base PostgreSQL cohérente et pour implémenter les mêmes relations dans Prisma. Grâce à cette démarche, ce qui est modélisé côté métier se retrouve bien dans la base réelle, et donc dans le comportement de l'application.

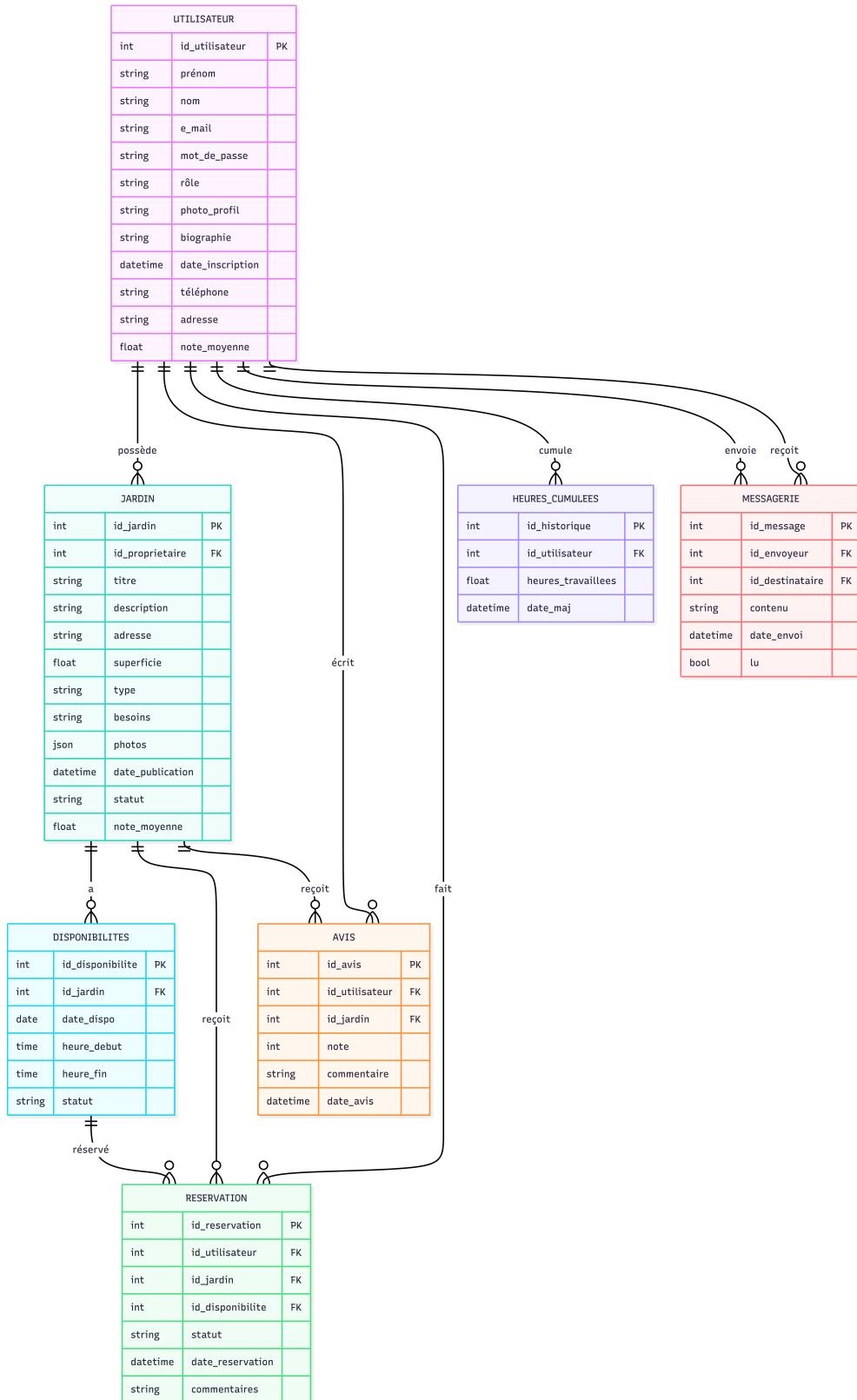


FIGURE 7.2 – MLD - traduction en tables, clés et relations (proche de l'implémentation).

Pourquoi PostgreSQL ?

J'ai choisi PostgreSQL parce que Jardin Solidaire manipule des données **structurées et fortement liées** : utilisateur.ices, jardins, créneaux, réservations, messages, avis... Une base relationnelle me permet de modéliser ces liens proprement (clés étrangères, contraintes), et donc d'éviter des incohérences dès l'écriture en base.

Ce choix est surtout motivé par la fonctionnalité la plus sensible du projet : la **réservation**. Quand un.e utilisateur.ice réserve, je dois garantir que l'opération est **fiable**, même si deux personnes agissent en même temps. Avec PostgreSQL, je peux m'appuyer sur des **transactions** et sur ses garanties **ACID** pour sécuriser ces écritures.

ACID décrit quatre garanties que j'utilise concrètement dans le projet :

- **Atomicité** : une action critique s'exécute entièrement ou pas du tout. Par exemple, lors d'une réservation, je veux éviter un état incohérent où la réservation est créée mais le créneau reste libre (ou l'inverse).
- **Cohérence** : les règles définies restent respectées. Par exemple, une réservation ne peut pas référencer un jardin inexistant, et les contraintes (formats, unicité, relations) restent valides après l'opération.
- **Isolation** : deux opérations simultanées n'interfèrent pas. C'est crucial si deux personnes tentent de réserver le même créneau au même moment : je veux une seule réservation valide, et un état final clair.
- **Durabilité** : une fois validée, l'écriture est persistée. Même en cas de redémarrage ou de panne, la donnée confirmée reste enregistrée.

Enfin, PostgreSQL s'intègre très bien avec **Prisma** et **Docker**. Ça me permet de travailler en local avec une base simple à lancer, et de garder une configuration proche de celle du déploiement : c'est plus reproductible et plus stable.

Chapitre 8

Spécifications techniques du projet

Choix technologiques

Pour construire Jardin Solidaire, j'ai choisi une stack web moderne, organisée en couches. Une **architecture en couches** signifie que l'on sépare clairement les responsabilités.

La première couche est l'**interface**, c'est-à-dire tout ce que l'utilisateur.icevoit et manipule dans son navigateur : les pages, les formulaires, les boutons, les listes, la carte, etc.

La deuxième couche est la **logique métier** : c'est le cerveau de l'application. Elle décide si une action est autorisée, applique les règles, vérifie les données et construit les réponses. Cette couche est portée par une **API**, c'est-à-dire un serveur qui reçoit des requêtes et renvoie des données.

La troisième couche est la **persistence des données** : c'est la **base de données**, l'endroit où l'on stocke durablement les informations (utilisateur.ice.s, jardins, créneaux, réservations, messages...).

Cette séparation est importante parce qu'elle rend le projet plus facile à faire évoluer. Si je change l'interface, je n'ai pas besoin de toucher aux règles métier. Cela facilite aussi les tests : on peut tester l'API sans lancer tout le front, et inversement. Enfin, cela rend la maintenance plus simple, car chaque partie a un rôle clair.



FIGURE 8.1 – Architecture en couches - séparation interface, logique métier et données.

Front-end : React / Next.js

J'ai développé le front-end avec **Next.js** (basé sur **React**) parce que l'application contient beaucoup d'écrans dynamiques : une liste de jardins avec filtres, des pages de détail, des profils, et des parcours de réservation. Construire l'interface en composants m'a permis de réutiliser des blocs d'UI (cartes, badges de statuts, formulaires) au lieu de dupliquer du code, ce qui réduit les incohérences et accélère les évolutions.

Next.js m'a aussi apporté une structure très claire grâce au routing par fichiers : les routes comme `/gardens` ou `/bookings/[id]` sont directement lisibles dans l'arborescence du projet. Enfin, j'ai choisi Next.js parce qu'il permet de **pré-rendre** certaines pages (côté serveur ou en statique), ce qui améliore la rapidité d'affichage et le **référencement (SEO)** si l'on souhaite rendre un jour certaines pages publiques. Couplé à son intégration avec **Vercel**, cela simplifie le déploiement et les mises à jour du front.

Back-end : Node.js / Express

Le back-end de Jardin Solidaire est une **API REST** développée en **Node.js** avec **Express**. J'ai choisi cette stack parce qu'elle est légère, très documentée, et qu'elle me permet de garder une structure simple et lisible : des **routes** (endpoints), des **contrôleurs** (traitement), et des **middlewares** (vérifications transverses).

J'ai exposé les fonctionnalités via des routes HTTP (par exemple pour lister des jardins ou créer une réservation) et j'ai renvoyé des réponses standardisées au front. Le back-end garantit la fiabilité : j'y ai centralisé l'authentification, les autorisations selon les rôles (jardinier.e / propriétaire) et les validations qui évitent les incohérences.

Pour rendre ces vérifications maintenables, j'ai utilisé des **middlewares** exécutés avant la logique métier, notamment pour vérifier l'authentification et configurer le **CORS**. Avec le CORS, j'ai limité les origines autorisées à appeler l'API afin d'éviter qu'une autre application **consomme l'API**.

Base de données : PostgreSQL

J'ai construit Jardin Solidaire autour d'une **base de données relationnelle**, parce que le cœur du projet repose sur des liens entre les données : un propriétaire possède des jardins, un jardin propose des créneaux, un.e jardinier.e réserve un créneau, et une réservation peut ensuite évoluer (statut, échanges, etc). Ce modèle relationnel me permet de garder une structure cohérente et d'éviter des données déconnectées.

J'ai choisi **PostgreSQL** parce que c'est un standard très courant en production et particulièrement adapté à des modèles relationnels. Il s'intègre très bien avec Prisma et l'écosystème Node.js, et il rend le projet plus facilement maintenable : si le projet est repris plus tard, PostgreSQL est familier dans la majorité des environnements professionnels.

ORM : Prisma

Pour connecter mon code à PostgreSQL, j'ai utilisé **Prisma** comme ORM (Object-Relational Mapping). Au lieu d'écrire du SQL à la main, j'ai manipulé des modèles directement dans le code (par exemple User, Garden, Booking), ce qui rend les requêtes plus lisibles et plus faciles à maintenir.

J'ai choisi Prisma parce qu'il m'a permis de garder le modèle de données **centralisé** et cohérent : toutes les relations et champs sont définis dans un schéma unique, et Prisma génère ensuite un client qui limite les erreurs (champs inexistant, formats incorrects, etc).

J'ai aussi utilisé Prisma pour gérer les **migrations**, afin de versionner l'évolution de la base de données (ajouts de tables, de champs, ajustements de relations) et pouvoir reproduire ces changements proprement sur une autre machine ou en production.

Enfin, en générant des requêtes paramétrées, Prisma réduit les risques liés aux injections SQL et m'aide à garder une couche d'accès aux données plus sûre.

Conteneurisation : Docker / Docker Compose

Pour le développement, j'ai utilisé **Docker** et **Docker Compose** afin d'avoir un environnement de travail reproductible. L'objectif était d'éviter les écarts entre machines (versions, dépendances, configuration).

J'ai packagé les services du projet et je les lance ensemble avec Docker Compose : le front-end, le back-end et la base de données PostgreSQL. Ça me permet de démarrer le projet rapidement, de le réinstaller facilement sur une nouvelle machine, et de garder une configuration stable tout au long du développement. C'est aussi une base saine pour un déploiement propre, puisque l'application est déjà pensée comme un ensemble de services lancables de manière cohérente.

Qualité : tests (Jest, Supertest, Playwright) et automatisation (GitHub Actions)

Pour fiabiliser Jardin Solidaire, j'ai mis en place une stratégie de tests à plusieurs niveaux. J'ai utilisé **Jest** pour écrire des **tests unitaires** sur des éléments clés (fonctions et règles métier), afin de sécuriser rapidement des comportements critiques et éviter des bugs lors des évolutions. J'ai aussi ajouté des **tests d'intégration** avec **Supertest** pour vérifier le comportement de l'API dans des conditions proches du réel. Cela me permet de tester l'API sans passer par l'interface, tout en restant sur des scénarios concrets. J'ai utilisé **Playwright** pour des **tests end-to-end**, afin de rejouer des parcours complets d'un.e utilisateur.ice : navigation, formulaires, appels à l'API et affichage des résultats. Je vérifie ainsi que l'application fonctionne correctement de bout en bout. Enfin, j'ai automatisé ces vérifications avec **GitHub Actions** : à chaque push ou pull request, je lance automatiquement les tests et je vérifie que l'application build correctement. L'objectif est de détecter les régressions le plus tôt possible et de ne pas dépendre uniquement de tests manuels.

Gestion de code : Git / GitHub

J'ai versionné le projet avec **Git** et je l'ai hébergé sur **GitHub** afin de garder un historique clair et de travailler de manière structurée. J'ai organisé mon développement avec des branches par fonctionnalité, ce qui m'a permis d'isoler les évolutions, de tester sans risque et de pouvoir revenir en arrière facilement en cas de problème. J'ai aussi utilisé GitHub pour cadrer mon workflow avec des **pull requests** et l'intégration de la CI, afin de valider automatiquement que le projet build et que les tests passent avant d'intégrer une évolution.

Organisation du projet

J'ai organisé le code de Jardin Solidaire pour séparer clairement les responsabilités et garder une base facile à faire évoluer. J'ai isolé l'interface dans **frontend/** (Next.js) et la logique métier dans **backend/** (Node/Express), afin d'éviter que des règles métier se retrouvent dispersées dans l'UI. J'ai aussi dédié un dossier **e2e/** aux tests end-to-end (Playwright) pour valider des parcours complets sans mélanger ces scénarios avec le code applicatif. Enfin, j'ai centralisé l'automatisation dans **.github/workflows/** et j'ai regroupé à la racine les configurations communes (Docker, Jest, Playwright...) ainsi que les fichiers **.env** pour gérer la configuration et les secrets hors du code et qui ne sont pas versionnés sur GitHub.



FIGURE 8.2 – Arborescence du projet - séparation front-end, back-end, tests et automatisation.

Flux technique : du navigateur à la base de données

J'ai formalisé le trajet d'une action d'un.e utilisateur.ice pour montrer où s'appliquent les règles et où circulent les données. Quand un.e utilisateur.ice déclenche une action (par exemple demander une réservation), le navigateur appelle le front **Next.js** (Vercel), qui envoie ensuite une requête à l'**API Express (Render)**. C'est côté API que j'applique l'authentification, les autorisations et les règles métier, avant d'interroger **PostgreSQL (Neon)** via **Prisma** et de renvoyer une réponse au front. Même si l'interface affiche un créneau comme disponible, le serveur doit revalider la disponibilité au moment de la demande afin d'éviter une double réservation.

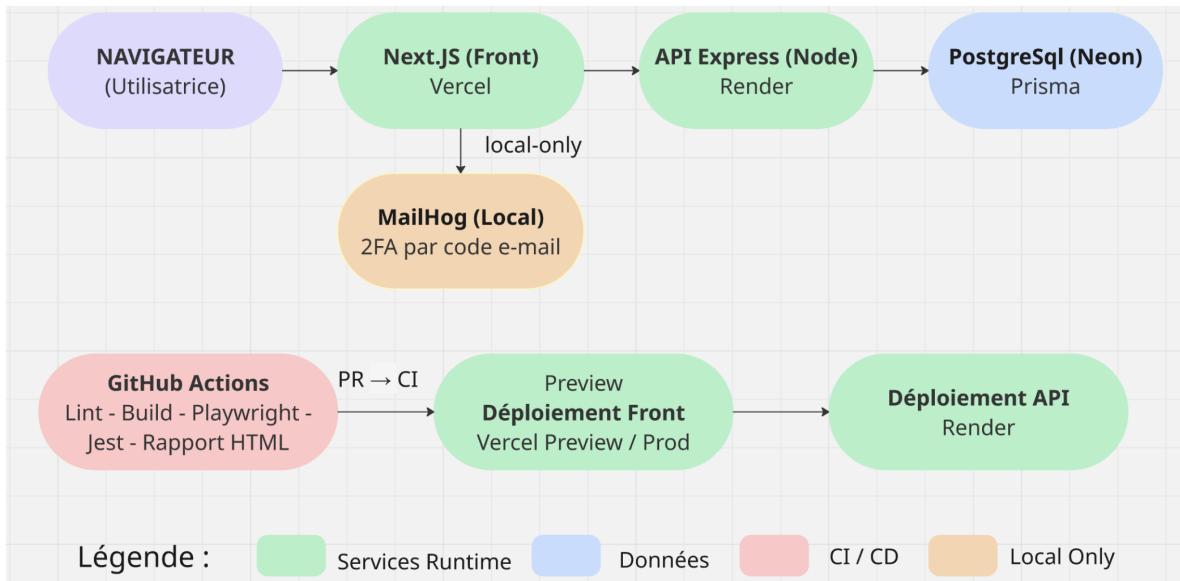


FIGURE 8.3 – Flux technique

Chapitre 9

Gestion de projet

Organisation et contexte

Jardin Solidaire a été lancé à trois, puis j'ai poursuivi le développement seule pour amener le MVP jusqu'au bout. En autonomie, j'ai gardé une organisation proche d'un contexte d'équipe, afin de suivre l'avancement de manière claire, de maintenir une base de code saine, et d'éviter les retours en arrière.

En amont, nous avons rédigé un cahier des charges interne qui a servi de fil conducteur : objectifs, périmètre du MVP, priorités, contraintes et planning.

Annexe C — Cahier des charges

Suivi : Kanban en flux continu et priorisation MoSCoW

Pour piloter l'avancement, j'ai utilisé un tableau **Kanban** en flux continu : j'ai fait avancer les tâches étape par étape (à faire → en cours → à relire/tester → terminé), sans découpage en sprints. Ce format correspondait au projet : je réajustais les priorités au fil des retours et des blocages techniques, tout en gardant une vision du prochain objectif.

Pour prioriser, j'ai appliqué la méthode **MoSCoW** afin de protéger le MVP : j'ai d'abord livré le *Must have* (parcours essentiel), puis seulement ensuite ce qui améliorait l'expérience sans être bloquant. J'ai découpé chaque fonctionnalité en tickets concrets et testables (profil, affichage des jardins, réservation d'un créneau...), ce qui m'a permis d'avancer par petites unités, de valider plus souvent, et de limiter le risque de tâches trop longues ou floues.

Workflow : du ticket à l'intégration

J'ai mis en place un workflow reproductible, du besoin jusqu'à l'intégration du code. Concrètement, je pars d'un ticket, je développe sur une branche dédiée, puis j'ouvre une pull request, ce qui permet une relecture à froid, une description claire du changement, et une traçabilité de l'évolution du projet. Couplé à la CI, ce workflow m'a permis d'intégrer plus sereinement et de limiter les régressions.

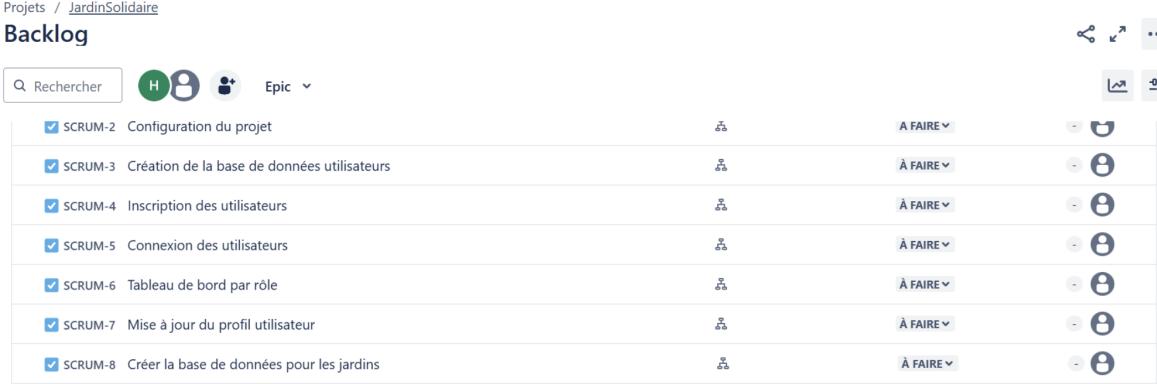
Ticket Jira → Branche Git (feat/... ou fix/...) → Commits → Pull Request → Merge sur Main

FIGURE 9.1 – Workflow : du ticket à l'intégration

Definition of Done

Pour éviter des tickets terminés mais incomplets, j'ai mis en place une **Definition of Done** simple :

- le code est développé sur une **branche dédiée** liée au ticket;
- la fonctionnalité est **testable** (cas nominal + principaux cas limites);
- la **CI est au vert**;
- la pull request est prête à être fusionnée (description claire et changements identifiables).



SCRUM-2 Configuration du projet	À FAIRE
SCRUM-3 Création de la base de données utilisateurs	À FAIRE
SCRUM-4 Inscription des utilisateurs	À FAIRE
SCRUM-5 Connexion des utilisateurs	À FAIRE
SCRUM-6 Tableau de bord par rôle	À FAIRE
SCRUM-7 Mise à jour du profil utilisateur	À FAIRE
SCRUM-8 Crérer la base de données pour les jardins	À FAIRE

FIGURE 9.2 – Backlog

Qualité et automatisation

Pour sécuriser l'intégration, j'ai connecté la **CI** avec **GitHub Actions**. À chaque push ou pull request, je lance automatiquement les vérifications principales (lint, tests unitaires, tests d'intégration, tests end-to-end Playwright). L'objectif est simple : détecter une erreur le plus tôt possible, garder une base de code stable, et éviter qu'un changement casse une fonctionnalité déjà validée.

Chapitre 10

Développement

Structure Front-End

Pour ce zoom technique, j'ai choisi de me concentrer sur la réservation, car c'est une **fonctionnalité clé** de Jardin Solidaire : c'est elle qui transforme une intention ("j'aimerais aider dans ce jardin") en action concrète ("j'ai un créneau validé, je sais quand venir, et je peux échanger").

J'ai développé le front avec **Next.js** et j'ai structuré ce **parcours critique** autour de deux pages : une vue d'ensemble, puis un détail. J'ai donc créé `/bookings/page.js` pour l'écran "Mes réservations" (liste sous forme de cartes avec jardin, date/heure et statut), et `/bookings/[id]` pour afficher le détail d'une réservation sélectionnée.

Comme on le voit sur la Figure 10.1, cette organisation me permet de séparer clairement ce qui relève de la logique de page (récupérer et afficher des données), de ce qui relève de l'interface (**composants réutilisables**). J'ai choisi ce découpage parce qu'il rend le code plus lisible, et parce qu'il facilite l'évolution du parcours sans multiplier la duplication.

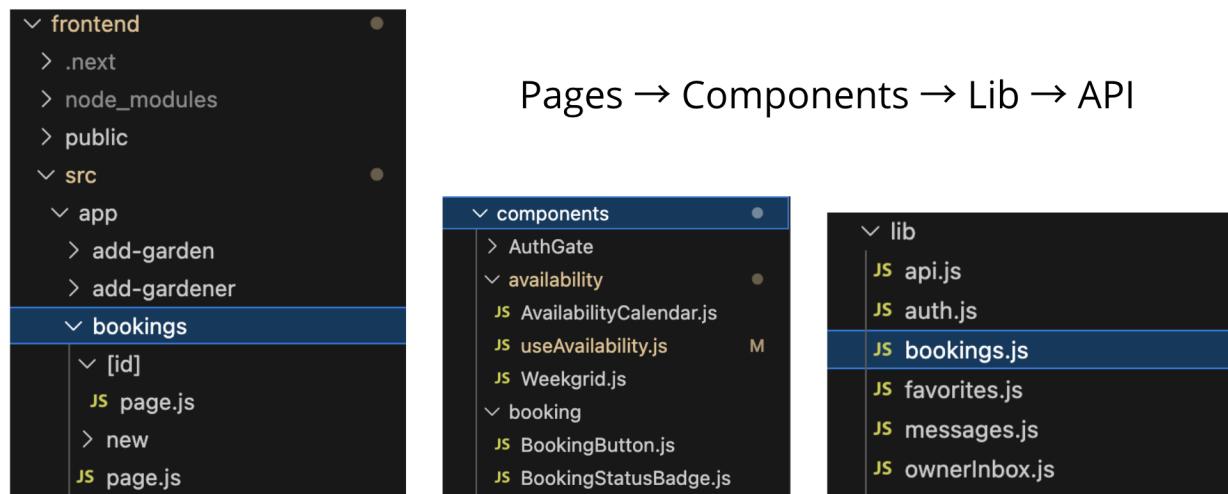


FIGURE 10.1 – Organisation front-end autour des pages `/bookings` et `/bookings/[id]` et des composants réutilisables.

Autour de ces pages, j'ai découpé l'interface en composants réutilisables (cartes, badges de statut, actions, planning) pour éviter la duplication et garder une UI cohérente. J'ai aussi centralisé la communication avec le back-end dans `lib/bookings.js` plutôt que de retrouver des `fetch` disséminés partout : je garde des appels homogènes, un code plus lisible, et une maintenance plus simple lorsque le parcours évolue.

Pages Front-End : liste et détail

Sur la page `/bookings/page.js`, j'ai fait en sorte que l'utilisateur.ice retrouve immédiatement ses informations. Au chargement, je récupère les réservations de l'utilisateur.ice connecté-e via l'API, puis je stocke le résultat dans le `state` pour afficher la liste. J'ai choisi d'expliciter les **états de chargement et d'erreur** afin que le comportement reste lisible : soit on voit que ça charge, soit on comprend pourquoi ça n'a pas abouti (Figure 10.2).

```
17 function BookingsListInner() {
18   const [items, setItems] = useState([]);
19   const [err, setErr] = useState('');
20   const sp = useSearchParams();
21   const router = useRouter();
22
23   useEffect(() => {
24     const token = localStorage.getItem('token');
25     if (!token) router.push('/login');
26   }, [router]);
27
28   useEffect(() => {
29     (async () => {
30       try {
31         const data = await getMyBookings();
32         setItems(Array.isArray(data) ? data : []);
33       } catch {
34         setErr('impossible de charger vos réservations');
35       }
36     })();
37   }, []);
}
```

FIGURE 10.2 – /bookings/page.js

Après avoir déclenché le fetch, j'ai structuré l'expérience de la page en suivant l'ordre réel vécu par l'utilisateur.ice. D'abord, j'affiche un **fallback de chargement** au niveau de la page, pour rendre l'attente explicite. Ensuite, si la réservation vient d'être créée, j'affiche un message de succès via le paramètre `created=1`. Enfin, si la réponse ne contient aucune réservation, je montre un état "liste vide". J'ai fait ce choix pour éviter les pages silencieuses : à chaque étape (chargement, confirmation, absence de données), l'interface explique clairement ce qu'il se passe (Figure 10.3).

```
9  export default function Page() {
10    return (
11      <Suspense fallback={<main className="max-w-5xl mx-auto p-6">chargement des réservations...</main>}>
12        <BookingsListInner />
13      </Suspense>
14    );
15  }

39  const justCreated = sp.get('created') === '1';
40

50
51  {justCreated && (
52    <div className="rounded-md bg-emerald-50 p-3 text-emerald-800">Réservation créée</div>
53  )}
54

55  {!items.length && !err && <li className="text-gray-500">Aucune réservation pour le moment.</li>}
56</ul>
```

FIGURE 10.3 – /bookings/page.js

Sur la page `/bookings/[id]`, j'ai utilisé une route dynamique pour accéder au détail d'une réservation. Je récupère l'identifiant dans l'URL via `useParams`, puis je charge la réservation correspondante et j'affiche les informations utiles (créneau, statut, actions disponibles). Ce choix permet à l'utilisateur.ice de passer de la vue liste à une vue **actionnable** (Figure 10.4).

```
14  export default function BookingDetailPage() {
15    const { id } = useParams();
16    const router = useRouter();
17
18    const [data, setData] = useState(null);
19    const [err, setErr] = useState('');
20    const [busy, setBusy] = useState(false);
21
22    useEffect(() => {
23      const token = localStorage.getItem('token');
24      if (!token) router.push('/login');
25    }, [router]);
26
27    useEffect(() => {
28      (async () => {
29        try {
30          const r = await getBooking(id);
31          setData(r);
32        } catch {
33          setErr('introuvable');
34        }
35      })();
36    }, [id]);
37
```

FIGURE 10.4 – `/bookings/[id]`

Enfin, quand l'utilisateur.ice déclenche une action, j'envoie une requête PATCH au back-end. Côté front, l'affichage dépend du state : c'est l'état React qui stocke la réservation courante et qui sert de **source de vérité** pour rendre l'interface (statut affiché, boutons disponibles, messages). J'attends la réponse serveur et je mets à jour le state uniquement avec la réservation renvoyée par l'API, ce qui évite les décalages (UI qui affiche "confirmée" alors que le serveur a refusé, conflit ou autorisation manquante).

Composants

J'ai conçu mes composants autour de deux objectifs : **sécuriser les accès et standardiser l'interface** pour limiter la duplication et garantir des comportements identiques partout.

Sécuriser les pages sensibles : AuthGate

D'abord, j'ai créé **AuthGate** pour protéger les pages qui ne doivent pas être accessibles sans session (ex. /my-space, réservations, favoris). J'ai fait ce choix pour éviter qu'un.e utilisateur.ice arrive sur un écran cassé (données absentes, erreurs de requêtes, page vide) : si la session n'est pas valide, l'accès doit être bloqué.

AuthGate s'appuie sur `useAuth()` pour lire l'état de session (`loading` et `isAuthenticated`). Tant que la vérification est en cours, j'affiche un feedback clair ("Vérification de la session..."). Une fois la session confirmée, j'autorise le rendu des `children`. Si la personne n'est pas authentifiée, je redirige vers /login en conservant une URL de retour (`next=/my-space`). J'ai choisi ce mécanisme pour garantir une expérience **fluide** (retour à l'endroit prévu) tout en gardant une logique **centralisée**.

```
frontend > src > components > AuthGate > JS page.js > ...
1  'use client';
2  import { useEffect } from 'react';
3  import { useRouter } from 'next/navigation';
4  import { useAuth } from '@/lib/useAuth';
5
6  export default function AuthGate({ children }) {
7    const { isAuthenticated, loading } = useAuth();
8    const router = useRouter();
9
10   useEffect(() => {
11     if (!loading && !isAuthenticated) {
12       router.replace('/login?next=/my-space');
13     }
14   }, [loading, isAuthenticated, router]);
15
16   if (loading) {
17     return <div className="pt-24 text-center text-gray-600">Vérification de la session...</div>;
18   }
19   if (!isAuthenticated) return null;
20
21   return children;
22 }
23
```

FIGURE 10.5 – AuthGate : vérification de session, feedback pendant le chargement, et redirection si non authentifié.

Standardiser une UI métier : GardenOwnerCard

Ensuite, j'ai conçu un composant métier réutilisable : **GardenOwnerCard**. L'idée est simple : dès que j'affiche un jardin côté propriétaire (ex. /my-gardens, tableau de bord, liste de gestion), je réutilise **le même rendu et les mêmes actions**. J'ai choisi ce composant parce qu'il cristallise une logique métier claire : un.e propriétaire doit pouvoir **modifier, retirer/supprimer ou consulter** une fiche jardin.

Je le réutilise dans plusieurs écrans de gestion côté propriétaire (liste, tableau de bord, vues de suivi), ce qui me permet de conserver les mêmes repères et le même comportement.

Centraliser ces actions dans une carte unique me permet :

- de garder une UI **homogène** (mêmes boutons, mêmes libellés, mêmes états),
- de réduire la duplication,
- et d'assurer une gestion d'erreur identique (mêmes endpoints, mêmes retours).

Dans ce composant, l'action "Retirer" dépend de l'état métier : si le jardin est publié (publishedAt), je déclenche un POST /unpublish. S'il est déjà en brouillon, je permets une suppression (DELETE). J'ai fait ce choix pour coller à la réalité produit : **retirer** un jardin visible n'a pas le même impact que **supprimer** un brouillon.

```
frontend > src > components > GardenOwnerCard > js GardenOwnerCard.js > GardenOwnerCard > [o] onRetire
  1 import { useRouter } from 'next/router';
  2 import { apiFetch } from '@lib/api';
  3
  4 export default function GardenOwnerCard({ garden, onChanged }) {
  5   const router = useRouter();
  6
  7   const onEdit = () => {
  8     router.push(`'/garden/${garden.id}/edit`);
  9   };
 10
 11   const onRetire = async () => {
 12     const path = garden.publishedAt ? `/gardens/${garden.id}/unpublish` : `/gardens/${garden.id}`;
 13     try {
 14       if (garden.publishedAt) {
 15         await apiFetch(path, { method: 'POST' });
 16       } else {
 17         await apiFetch(path, { method: 'DELETE' });
 18       }
 19     } catch (e) {
 20       onChanged?.();
 21       alert(e?.message || 'Échec de la mise à jour.');
 22     }
 23   };
 24
 25   const onView = () => router.push(`'/garden/${garden.id}`);
 26
 27   return (

```

FIGURE 10.6 – GardenOwnerCard : actions standardisées (edit/unpublish/delete/view) et rafraîchissement via onChanged.

Réservations : UI cohérente + logique partagée (statut + actions)

Sur le parcours réservation, j'ai appliqué la même approche : des composants partagés et un flux identique entre la liste et le détail. Mon objectif est que l'utilisateur voie immédiatement où en est sa demande et que l'interface reflète toujours l'état réel de la réservation.

Un même composant réutilisé : BookingStatusBadge

J'ai isolé l'affichage du statut dans BookingStatusBadge pour garantir le même rendu sur la page liste et sur la page détail. J'ai fait ce choix parce que le statut est un repère central : il indique immédiatement où en est la demande (*en attente*, *confirmée*, *annulée*, *terminée*). En le centralisant, j'évite de récrire la logique à plusieurs endroits et je simplifie la maintenance : si je change une règle d'affichage, tout reste cohérent (Figure 10.7).

```
 1  'use client';
 2
 3  import React from 'react';
 4
 5  const colors = {
 6    pending: 'bg-yellow-100 text-yellow-800',
 7    confirmed: 'bg-blue-100 text-blue-800',
 8    cancelled: 'bg-red-100 text-red-800',
 9    completed: 'bg-green-100 text-green-800',
10  };
11
12  export default function BookingStatusBadge({ status }) {
13    const cls = colors[status] || 'bg-gray-100 text-gray-800';
14    return (
15      <span className={` inline-block rounded-full px-3 py-1 text-sm font-medium ${cls}`}>
16        {status}
17      </span>
18    );
19  }

```

FIGURE 10.7 – BookingStatusBadge : composant partagé qui rend le statut de manière identique sur la liste et sur le détail.

Actions : un handler commun piloté par le state

Sur la page détail /bookings/[id], l'utilisateur.ice peut déclencher plusieurs actions (*confirmer, annuler, terminer*). Plutôt que trois logiques différentes, j'ai regroupé ces actions dans une configuration unique, exécutée via un **handler** commun.

Un **handler** est une fonction déclenchée lors d'un événement utilisateur.ice (ici : un clic). J'ai choisi ce modèle pour garantir le même comportement pour chaque bouton : même gestion de chargement, mêmes retours, et même mise à jour du rendu. Concrètement, chaque bouton appelle le **même** handler (`doAction`) avec un paramètre différent (Figure 10.8).

```
/  
8  const ACTIONS = [  
9    { key: 'confirm', label: 'Confirmer', patch: { status: 'confirmed' } },  
10   { key: 'cancel', label: 'Annuler', patch: { status: 'cancelled' } },  
11   { key: 'complete', label: 'Marquer terminé', patch: { status: 'completed' } },  
12 ];  
13
```

FIGURE 10.8 – /bookings/[id] : plusieurs boutons, mais une configuration unique et un déclenchement via un handler partagé.

Le flux UI reste simple : **clic utilisateur.ice → appel API → réponse serveur → mise à jour du state → rafraîchissement de l'interface.**

Ici, le **state** correspond aux **données en mémoire** utilisées pour afficher l'écran (par exemple : la réservation et son statut). Je ne mets à jour ce state **qu'après validation par l'API** : tant que le serveur n'a pas confirmé, je n'affiche pas un état supposé. J'ai fait ce choix pour que le rendu reste **fiable** : l'utilisateur.ice voit uniquement un état confirmé par le back-end, même en cas de latence ou d'erreur.

```
38  async function doAction(patch) {  
39    try {  
40      setBusy(true);  
41      const updated = await updateBooking(id, patch);  
42      setData(updated);  
43    } catch (e) {  
44      setErr(e.message || 'erreur');  
45    } finally {  
46      setBusy(false);  
47    }  
48 }
```

FIGURE 10.9 – `doAction` : handler partagé qui pilote l'appel API, l'état `busy` et la mise à jour du **state** (donc du rendu UI).

En procédant ainsi, je garde le front **léger** : il gère l'affichage et les interactions, tandis que les règles métier (conflits, cohérence, rôles et autorisations) restent côté back-end.

Accessibilité : formulaires compréhensibles et utilisables

Sur /bookings/new, j'ai appliqué des règles simples d'accessibilité pour rendre le formulaire compréhensible et utilisable : chaque champ possède un **label associé**, les erreurs sont affichées en clair, et un feedback visuel indique quand un créneau est indisponible. J'ai choisi ces éléments parce qu'ils réduisent directement les erreurs et évitent les situations où l'utilisateur.ice ne comprend pas pourquoi la validation échoue.

```
15  function NewBookingInner() {
85    <form onSubmit={handleSubmit} className="space-y-4">
86      <label className="block">
87        <span className="block text-sm mb-1">Jardin (ID)</span>
88        <input
89          value={gardenId}
90          onChange={(e) => {
91            setGardenId(e.target.value);
92            runCheck(startsAt, endsAt, e.target.value);
93          }}
94          className="w-full rounded border p-2"
95          placeholder="ex: 42"
96        />
97      </label>
98
99      <label className="block">
100        <span className="block text-sm mb-1">Titre (optionnel)</span>
101        <input
102          value={title}
103          onChange={(e) => setTitle(e.target.value)}
104          className="w-full rounded border p-2"
105          placeholder="ex: tonte + taille"
106        />
107      </label>
108
109      <div className="grid grid-cols-1 md:grid-cols-2 gap-4">
110        <label className="block">
111          <span className="block text-sm mb-1">Début</span>
112          <input
113            type="datetime-local"
114            value={startsAt}
115            onChange={(e) => {
116              setStartsAt(e.target.value);
117              runCheck(e.target.value, endsAt, gardenId);
118            }}
119            className="w-full rounded border p-2"
120          />
121      </label>
```

FIGURE 10.10 – /bookings/new - Labels + gestion d'erreurs + feedback de disponibilité.

Pour cadrer cette démarche, je me suis appuyée sur une checklist inspirée des principes RGAA :

- champs de formulaire avec **label associé** (et aide de saisie si besoin),
- **navigation clavier** possible sur les éléments interactifs,
- **focus visible** sur boutons et champs actifs,
- textes et boutons avec un **contraste suffisant**,
- images porteuses d'information avec attribut alt,
- erreurs affichées de manière **compréhensible** et actionnable.

Ces points constituent une base d'amélioration continue : l'objectif est d'éviter qu'une fonctionnalité critique (réserver / se connecter) devienne bloquante pour certain·es utilisateur·ices. Je n'ai pas encore une conformité RGAA complète à ce stade, mais j'ai appliqué ces critères de base sur les parcours critiques et posé une checklist pour continuer à améliorer l'accessibilité au fil des itérations.

Connexion au Back-End

Pour éviter de disperser des `fetch` et garantir un comportement constant, j'ai **centralisé** les appels à l'API dans `lib/bookings.js`. J'ai choisi ce point d'entrée unique parce qu'il me permet de réutiliser la même logique partout : si l'API évolue (URL, headers, format d'erreur), je n'ai qu'un seul fichier à maintenir (Figure 10.11).

Au-delà de la centralisation, j'ai fait ce choix pour respecter une séparation claire des responsabilités : l'UI n'a pas vocation à gérer les détails réseau (construction d'URL, headers, parsing, gestion d'erreurs). Le rôle des pages est d'exprimer une intention ("charger mes réservations", "créer une réservation"), tandis qu'une **couche d'accès API** encapsule les appels HTTP. Cette séparation rend le code plus lisible, plus testable, et évite que chaque écran réimplémente sa propre version des mêmes règles.

```
1  const API_BASE = process.env.NEXT_PUBLIC_API_URL || 'http://localhost:5001';
2
3  function authHeaders() {
4    const token = typeof window !== 'undefined' ? localStorage.getItem('token') : null;
5    return token ? { Authorization: `Bearer ${token}` } : {};
6  }
7
8  async function handle(res, fallbackError = 'request_failed') {
9    const data = await res.json().catch(() => ({}));
10   if (!res.ok) throw new Error(data?.error || data?.reasons?.[0] || fallbackError);
11   return data;
12 }
```

FIGURE 10.11 – `lib/bookings.js` : couche d'accès API centralisée pour le domaine "bookings".

Chaque action métier passe ensuite par une fonction dédiée (`createBooking`, `getMyBookings`, `getBooking`, `updateBooking`, etc). J'ai fait ce choix pour pouvoir appeler clairement l'intention depuis les pages, sans mélanger UI et détails réseau, et pour garder un point d'entrée homogène quel que soit l'écran (Figure 10.12).

```
14  export async function createBooking(payload) {
15    const res = await fetch(`${API_BASE}/api/bookings`, {
16      method: 'POST',
17      headers: { 'Content-Type': 'application/json', ...authHeaders() },
18      body: JSON.stringify(payload),
19    });
20    return handle(res, 'create_failed');
21  }
22
23  export async function getMyBookings() {
24    const res = await fetch(`${API_BASE}/api/bookings/me`, {
25      headers: { 'Content-Type': 'application/json', ...authHeaders() },
26      cache: 'no-store',
27    });
28    return handle(res, 'list_failed');
29  }
30
31  export async function getBooking(id) {
32    const res = await fetch(`${API_BASE}/api/bookings/${id}`, {
33      headers: { 'Content-Type': 'application/json', ...authHeaders() },
34      cache: 'no-store',
35    });
36    return handle(res, 'not_found');
37  }
38
39  export async function updateBooking(id, patch) {
40    const res = await fetch(`${API_BASE}/api/bookings/${id}`, {
41      method: 'PATCH',
42      headers: { 'Content-Type': 'application/json', ...authHeaders() },
43      body: JSON.stringify(patch),
44    });
45    return handle(res, 'update_failed');
46  }
```

FIGURE 10.12 – Fonctions dédiées par action : l'UI appelle une intention, l'API layer gère le réseau.

J'ai aussi créé des **fonctions utilitaires** pour rendre les appels homogènes. Avec `authHeaders()`, j'ajoute automatiquement le token JWT dans le header `Authorization` au format `Bearer <token>`. J'ai choisi ce mécanisme pour éviter les oubli et sécuriser l'accès aux routes protégées (Figure 10.11).

Avec `handle(res)`, je centralise le traitement des réponses : lecture du contenu, parsing quand c'est nécessaire, et normalisation des erreurs pour renvoyer un format exploitable par l'interface. L'objectif est que l'utilisateur.ice reçoive un retour clair quel que soit l'écran, tout en gardant le front **léger** (Figure 10.11).

```

48  export async function canBook({ gardenId, startsAt, endsAt }) {
49    const params = new URLSearchParams({
50      gardenId: String(gardenId),
51      startsAt: new Date(startsAt).toISOString(),
52      endsAt: new Date(endsAt).toISOString(),
53    });
54    const res = await fetch(`${API_BASE}/api/bookings/can-book?${params.toString()}`, {
55      headers: { ...authHeaders() },
56      cache: 'no-store',
57    });
58    return handle(res, 'check_failed');
59  }

```

FIGURE 10.13 – `canBook` : exemple d'appel centralisé pour vérifier côté serveur qu'un créneau est réservable.

Enfin, dans cette couche d'accès API, j'ai **regroupé** et **nommé explicitement** les appels liés aux actions "propriétaire" (fonctions préfixées `owner*` comme `getOwnerInbox`, `ownerConfirmBooking`, `ownerCancelBooking`). J'ai fait ce choix pour rendre la lecture immédiate : on identifie tout de suite quelles fonctions correspondent aux actions "owner", et on évite de mélanger des intentions et des permissions différentes dans les pages (Figure 10.14).

```

61  export async function getOwnerInbox() {
62    const res = await fetch(`${API_BASE}/api/bookings/inbox`, {
63      headers: { 'Content-Type': 'application/json', ...authHeaders() },
64      cache: 'no-store',
65    });
66    return handle(res, 'inbox_failed');
67  }
68
69  export async function ownerConfirmBooking(id) {
70    const res = await fetch(`${API_BASE}/api/bookings/${id}/confirm`, {
71      method: 'POST',
72      headers: { 'Content-Type': 'application/json', ...authHeaders() },
73    });
74    return handle(res, 'confirm_failed');
75  }
76
77  export async function ownerCancelBooking(id) {
78    const res = await fetch(`${API_BASE}/api/bookings/${id}/cancel`, {
79      method: 'POST',
80      headers: { 'Content-Type': 'application/json', ...authHeaders() },
81    });
82    return handle(res, 'cancel_failed');
83  }

```

FIGURE 10.14 – Extraits de `lib/bookings.js` : fonctions dédiées aux actions côté propriétaire (`/inbox`, `/:id/confirm`, `/:id/cancel`), avec headers d'authentification et traitement de réponse centralisé.

Structure Back-End

Pour le back-end, j'ai structuré la réservation autour de deux ensembles de routes, parce que je voulais séparer clairement la demande de réservation et la gestion des créneaux. Comme on le voit sur Figure 10.15, j'ai regroupé tout ce qui concerne les réservations dans `routes/bookings.js` (création, liste mes réservations, consultation, mise à jour de statut et actions côté propriétaire comme confirmer ou annuler). J'ai isolé la gestion des disponibilités dans `routes/availability.js`, pour garder une logique lisible : un créneau se crée et se consulte indépendamment, puis une réservation vient s'y rattacher.

Pour sécuriser l'API, j'ai protégé les routes sensibles avec un middleware d'authentification (`requireAuth`). Il lit le token JWT, vérifie qu'il est valide, puis attache l'utilisateur.ice à la requête (`req.user`). Ensuite, dans les contrôleurs, je peux appliquer les autorisations et les règles métier côté serveur, sans dépendre de ce que le front envoie (voir Figure 9.14).

Route → `requireAuth` → controller → Prisma/DB → response

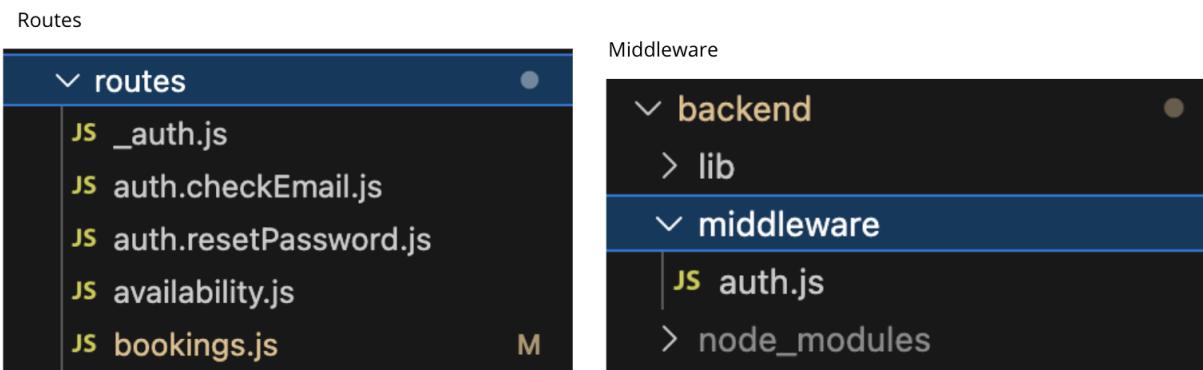


FIGURE 10.15 – Organisation back-end.

Contrôleur de création

Pour rendre la création de réservation fiable, j'ai structuré le back-end en couches : **routes → middleware d'authentification → contrôleurs → fonctions utilitaires → accès aux données Prisma.**

L'objectif est que chaque responsabilité soit au bon endroit : la route oriente la requête, le middleware sécurise, et le contrôleur exécute les opérations CRUD sur l'entité concernée. Ici, je me concentre sur l'entité Booking : la route POST /api/bookings redirige vers un contrôleur dédié à la création. L'accès est **protégé en amont** par le middleware requireAuth, qui vérifie le JWT et attache l'utilisateur.ice à req.user. Ensuite seulement, le contrôleur applique l'enchaînement métier : **valider l'entrée → vérifier l'absence de conflit → enregistrer.**

Cette route se décline en deux modes selon le payload : réservation d'un slot existant (slotId) ou création d'une réservation à partir d'une plage horaire (gardenId,startsAt/endsAt) (Figure 10.16).

```
209 // POST /api/bookings
210 router.post('/', requireAuth, async (req, res) => {
211   try {
212     const { slotId, gardenId, startsAt, endsAt, notes } = req.body || {};
213 
214     // Mode 1: book existing slot
215     if (slotId) {
216 
217       // Mode 2: free-dates -> create slot then book
218       if (!gardenId || !startsAt || !endsAt) {
```

FIGURE 10.16 – POST /api/bookings

Mode 1 - Réserver un slot existant

Dans le premier cas, l'utilisateur.ice réserve un créneau existant : le contrôleur vérifie que le slot existe, qu'il est encore disponible (status=free), puis contrôle qu'il ne chevauche pas une réservation déjà active. Si un conflit est détecté, je renvoie un 409 Conflict : cela indique clairement au front qu'on ne peut pas créer la réservation car l'état en base n'est plus compatible (Figure 10.17).

```
const slot = await prisma.availabilitySlot.findUnique({
  where: { id: BigInt(slotId) },
  select: { id: true, gardenId: true, status: true, date: true, startTime: true, endTime: true,
});
```

if (!slot) return res.status(404).json({ error: "slot_not_found" });
if (slot.status && slot.status !== "free") return res.status(400).json({ error: "slot_not_free" });

const { startsAt: requestedStartAt, endsAt: requestedEndAt } = buildRangeFromSlot(slot);
const conflict = await hasConflict({ gardenId: slot.gardenId, startsAt: requestedStartAt, endsAt: requestedEndAt });
if (conflict) return res.status(409).json({ error: "time_conflict" });

FIGURE 10.17 – Création (mode slotId) : validations + détection de conflit → 409 Conflict.

Mode 2 - Proposer une plage horaire

Dans le second cas, l'utilisateur.ice propose une plage horaire : je valide la cohérence des dates (début < fin, pas dans le passé), puis j'applique le même contrôle de chevauchement avant d'autoriser la création, pour empêcher qu'un créneau soit créé si une réservation active existe déjà sur la même plage (Figure 10.18).

```
const requestedStartAt = new Date(startsAt);
const requestedEndAt = new Date(endsAt);

if (isNaN(requestedStartAt.getTime()) || isNaN(requestedEndAt.getTime())) return res.status(400).json({ error: "invalid_dates" });
if (requestedStartAt >= requestedEndAt) return res.status(400).json({ error: "start_must_be_before_end" });
if (requestedEndAt < new Date()) return res.status(400).json({ error: "slot_in_past" });

const conflict = await hasConflict({ gardenId: Number(gardenId), startsAt: requestedStartAt, endsAt: requestedEndAt });
if (conflict) return res.status(409).json({ error: "time_conflict" });
```

FIGURE 10.18 – Création (mode gardenId/startAt/endsAt) : validations + contrôle de conflit avant insertion.

Fonctions utilitaires - Détection de chevauchement

La détection de chevauchement est isolée dans des **fonctions utilitaires** : elles prennent des plages horaires en entrée et renvoient un booléen (conflit / pas de conflit). Je convertis un slot en plage horaire (à partir de `date + startTime/endTime`), puis j'applique un test de recouvrement. Ce choix me permet de garder un comportement identique sur les deux modes de création et d'éviter de dupliquer des conditions dans les contrôleurs (Figure 10.19).

```
/*
 * -----
 * Helpers
 * -----
 */

function buildRangeFromSlot(slot) {
  if (!slot?.date || !slot?.startTime || !slot?.endTime) {
    return { startsAt: null, endsAt: null };
  }

  const slotDate = new Date(slot.date);

  const startTime = new Date(slot.startTime);
  const startsAt = new Date(slotDate);
  startsAt.setHours(
    startTime.getHours(),
    startTime.getMinutes(),
    startTime.getSeconds(),
    startTime.getMilliseconds()
  );

  const endTime = new Date(slot.endTime);
  const endsAt = new Date(slotDate);
  endsAt.setHours(
    endTime.getHours(),
    endTime.getMinutes(),
    endTime.getSeconds(),
    endTime.getMilliseconds()
  );

  return { startsAt, endsAt };
}

function rangesOverlap(rangeAStart, rangeAEnd, rangeBStart, rangeBEnd) {
  return !(rangeBEnd <= rangeAStart || rangeBStart >= rangeAEnd);
}

async function listDayBookings(gardenId, day) {
  const dayDate = new Date(day.toDateString());

  return prisma.booking.findMany({
    where: {
      gardenId: BigInt(gardenId),
      status: { in: ["pending", "confirmed"] },
      slot: { is: { date: dayDate } },
    },
    include: { slot: true },
  });
}

async function hasConflict({ gardenId, startsAt, endsAt, ignoreBookingId = null }) {
  if (!(startsAt instanceof Date) || isNaN(startsAt)) return null;
  if (!(endsAt instanceof Date) || isNaN(endsAt)) return null;

  const bookingsOfDay = await listDayBookings(gardenId, startsAt);

  for (const booking of bookingsOfDay) {
    if (ignoreBookingId && String(booking.id) === String(ignoreBookingId)) continue;

    const { startsAt: existingStartsAt, endsAt: existingEndsAt } =
      buildRangeFromSlot(booking.slot);
    if (existingStartsAt && existingEndsAt) {
      const overlap = rangesOverlap(
        startsAt,
        endsAt,
        existingStartsAt,
        existingEndsAt
      );
      if (overlap) {
        return { id: booking.id, slot: booking.slot, status: booking.status };
      }
    }
  }

  return null;
}
```

FIGURE 10.19 – Fonctions utilitaires anti-chevauchement : conversion slot → plage horaire (`buildRangeFromSlot`), test de recouvrement (`rangesOverlap`) et détection de conflit (fonction dédiée).

Transactions Prisma - garantir l'atomicité

Dans les deux modes, l'écriture en base passe par une **transaction Prisma**. Une transaction garantit l'**atomicité** : soit toutes les opérations s'exécutent, soit aucune n'est appliquée. J'ai fait ce choix pour éviter les états incohérents (par exemple : un slot marqué booked sans réservation créée, ou l'inverse), notamment en cas d'erreur ou de requêtes concurrentes.

Dans le mode `slotId`, je mets à jour le slot puis je crée la réservation dans la même transaction (Figure 10.20).

```
const created = await prisma.$transaction(async (tx) => {
  await tx.availabilitySlot.update({ where: { id: slot.id }, data: { status: 'booked' } });
  const booking = await tx.booking.create({
    data: {
      userId: BigInt(req.user.id),
      gardenId: slot.gardenId,
      slotId: slot.id,
      status: 'pending',
      notes: notes || null,
    },
    include: { slot: true, garden: { select: { id: true, title: true, ownerUserId: true } } },
  });

  if (booking.garden?.ownerUserId) {
    const { startsAt: s2, endsAt: e2 } = buildRangeFromSlot(booking.slot);
    const content = `Nouvelle demande de réservation pour "${booking.garden.title || 'Jardin'}" le ${s2?.toLocaleString()} → ${e2?.toLocaleString()}.`;
    await sendMessage({ fromUserId: req.user.id, toUserId: Number(booking.garden.ownerUserId), content });
  }

  return booking;
});
```

FIGURE 10.20 – Transaction Prisma (mode `slotId`) : mise à jour du slot (`status=booked`) puis création de la réservation (`status=pending`).

Dans le mode `gardenId/startAt/endsAt`, je crée d'abord le slot, puis la réservation associée, toujours dans une transaction unique (Figure 10.21).

```
const created = await prisma.$transaction(async (tx) => {
  const slot = await tx.availabilitySlot.create({
    data: {
      gardenId: BigInt(gardenId),
      date: dateOnly,
      startTime: s,
      endTime: e,
      status: 'booked',
    },
  });

  const booking = await tx.booking.create({
    data: {
      userId: BigInt(req.user.id),
      gardenId: BigInt(gardenId),
      slotId: slot.id,
      status: 'pending',
      notes: notes || null,
    },
    include: { slot: true, garden: { select: { id: true, title: true, ownerUserId: true } } },
  });

  if (booking.garden?.ownerUserId) {
    const { startsAt: s2, endsAt: e2 } = buildRangeFromSlot(booking.slot);
    const content = `Nouvelle demande de réservation pour "${booking.garden.title || 'Jardin'}" le ${s2?.toLocaleString()} → ${e2?.toLocaleString()}.`;
    await sendMessage({ fromUserId: req.user.id, toUserId: Number(booking.garden.ownerUserId), content });
  }

  return booking;
});
```

FIGURE 10.21 – Transaction Prisma (mode `gardenId/startAt/endsAt`) : création du slot puis de la réservation associée.

Modélisation des données (schéma Prisma)

J'ai traduit la réservation dans le schéma Prisma en séparant deux entités : le créneau (`AvailabilitySlot`) et la réservation (`Booking`). Ce découpage évite de mélanger offre et demande dans une seule table, ce qui rend les relations plus lisibles et les règles côté serveur plus simples à appliquer.

`AvailabilitySlot` représente l'offre : un créneau appartient à un jardin, décrit une plage horaire (date + heure de début/fin) et porte un statut (`free/booked/unavailable`). `Booking` représente la demande : il relie un-e utilisateur.ice à un jardin et à un créneau, avec un statut de suivi (en attente, confirmé, annulé, terminé) et, si besoin, un message (`notes`). Cette séparation apparaît dans le schéma Prisma : un slot peut exister indépendamment, et une réservation vient s'y associer lorsqu'une demande est faite (Figure 10.22).

```
/**  
 * -----  
 * Availability slots (disponibilites)  
 * -----  
 */  
model AvailabilitySlot {  
    id      BigInt    @id @default(autoincrement()) @map("id_disponibilite")  
    gardenId BigInt?  @map("id_jardin")  
    date    DateTime? @map("date_dispo") @db.Date  
    startTime DateTime? @map("heure_debut") @db.Time(6)  
    endTime  DateTime? @map("heure_fin") @db.Time(6)  
    status   String?   @map("statut")  
  
    garden  Garden?   @relation(fields: [gardenId], references: [id], onDelete: NoAction, onUpdate: NoAction)  
    bookings Booking[]  
  
    @@map("disponibilites")  
}  
  
/**  
 * -----  
 * Bookings (reservation)  
 * -----  
 */  
model Booking {  
    id      BigInt    @id @default(autoincrement()) @map("id_reservation")  
    userId  BigInt?  @map("id_utilisateur")  
    gardenId BigInt?  @map("id_jardin")  
    slotId  BigInt?  @map("id_disponibilite")  
    status   String?   @map("statut")  
    bookedAt DateTime? @default(now()) @map("date_reservation") @db.Timestamptz(6)  
    notes   String?   @map("commentaires")  
  
    slot  AvailabilitySlot? @relation(fields: [slotId], references: [id], onDelete: NoAction, onUpdate: NoAction)  
    garden Garden?         @relation(fields: [gardenId], references: [id], onDelete: NoAction, onUpdate: NoAction)  
    user   User?          @relation(fields: [userId], references: [id], onDelete: NoAction, onUpdate: NoAction)  
  
    @@map("reservation")  
}
```

FIGURE 10.22 – Modèles Prisma `AvailabilitySlot` et `Booking`.

Ce modèle sert à deux niveaux. Côté interface, il simplifie les affichages : je peux lister les créneaux disponibles sans mélanger l'historique des demandes, et afficher "mes réservations" comme un suivi de statuts et d'actions. Côté serveur, il clarifie l'intention : je crée une réservation uniquement si le créneau est encore disponible et compatible au moment de l'écriture en base, ce qui limite les doublons et maintient un état cohérent.

Sécurité et autorisations

La sécurité de l'API repose sur plusieurs mécanismes complémentaires : **protéger les identités, authentifier les requêtes**, puis **autoriser chaque action** côté serveur. L'objectif est que les contrôles critiques ne dépendent jamais de l'interface : n'importe qui peut appeler l'API directement (ex. via un script ou curl), donc le serveur doit rester la **source de vérité**.

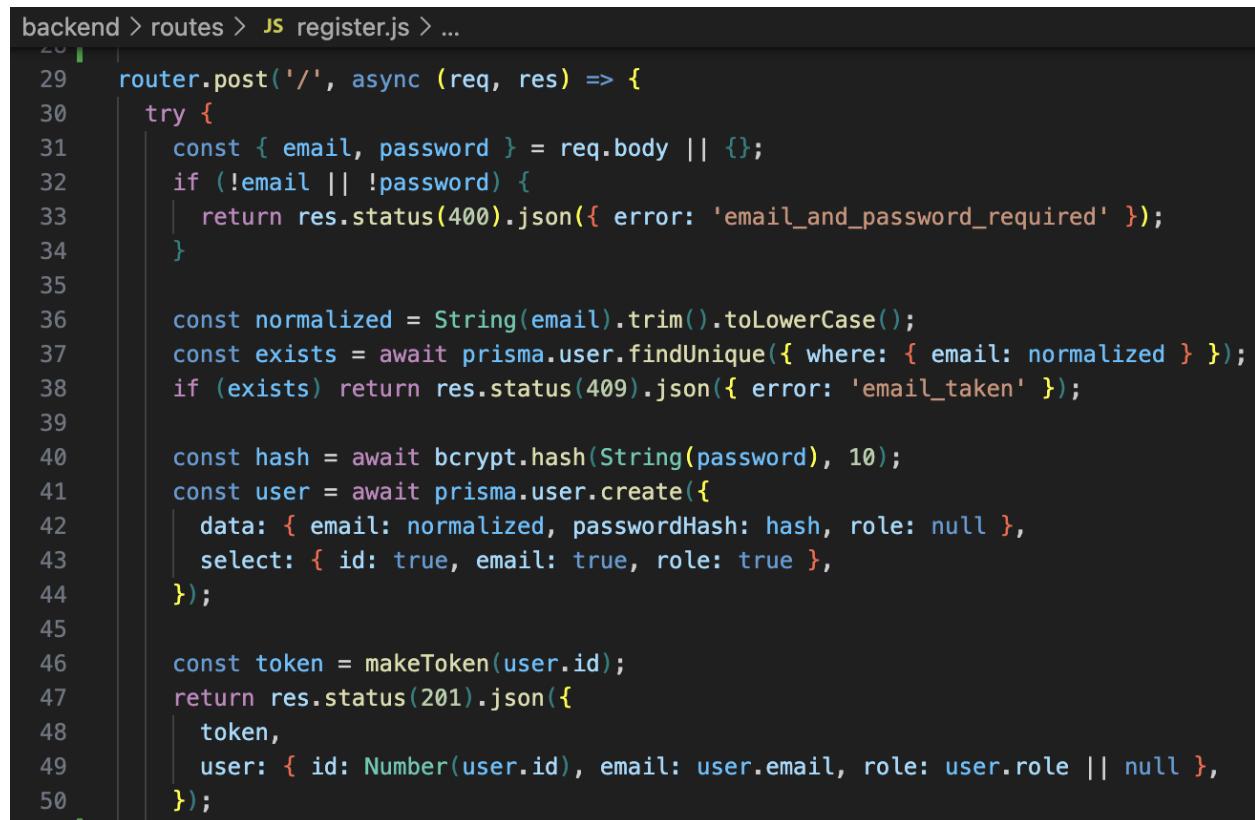
Protection des données personnelles (RGPD)

Dans Jardin Solidaire, j'ai identifié les obligations liées au RGPD dès la conception : l'application traite des données personnelles mais uniquement celles qui sont nécessaires au service. Concrètement, les informations manipulées concernent l'identité de compte (ex. email), le profil et le rôle (propriétaire / jardinier.e), ainsi que les données liées aux réservations (créneaux, statut, messages éventuels).

La base légale du traitement repose sur la création d'un compte et l'exécution du service demandé (gérer un jardin, réserver un créneau). J'ai appliqué un principe de **minimisation** : je ne collecte pas de données non indispensables au parcours, et je garde les informations strictement utiles au fonctionnement.

Protéger les identifiants : mot de passe haché

Pour protéger les comptes, je n'enregistre jamais un mot de passe en clair. Lors de la création de compte, je hash le mot de passe avec bcrypt : le hash est calculé dans un sens et ne peut pas être inversé, ce qui limite l'impact en cas de fuite de base de données. Lors de la connexion, je compare simplement le mot de passe saisi au hash stocké via bcrypt.compare.



```
backend > routes > JS register.js > ...
29   router.post('/', async (req, res) => {
30     try {
31       const { email, password } = req.body || {};
32       if (!email || !password) {
33         return res.status(400).json({ error: 'email_and_password_required' });
34       }
35
36       const normalized = String(email).trim().toLowerCase();
37       const exists = await prisma.user.findUnique({ where: { email: normalized } });
38       if (exists) return res.status(409).json({ error: 'email_taken' });
39
40       const hash = await bcrypt.hash(String(password), 10);
41       const user = await prisma.user.create({
42         data: { email: normalized, passwordHash: hash, role: null },
43         select: { id: true, email: true, role: true },
44       });
45
46       const token = makeToken(user.id);
47       return res.status(201).json({
48         token,
49         user: { id: Number(user.id), email: user.email, role: user.role || null },
50       });
51     }
52   }
53 }
```

FIGURE 10.23 – Inscription : hachage du mot de passe avec bcrypt.hash puis stockage en base dans passwordHash

Authentifier : JWT émis à la connexion, puis vérifié à chaque requête

Une fois les identifiants validés à la connexion, le serveur renvoie un JWT (JSON Web Token). Ce token est signé : il n'est pas falsifiable sans la clé serveur. Pour les routes protégées, l'interface envoie ensuite ce token dans le header `Authorization` au format `Bearer <token>`.

À chaque requête protégée, un middleware lit ce header, vérifie la signature du token et récupère l'utilisateur authentifié via `req.user`. Si le token est absent, mal formé ou invalide, je renvoie `401 Unauthorized` et la route ne s'exécute pas : la requête est bloquée **avant** d'arriver au contrôleur (Figure 10.24).

```
1 const jwt = require('jsonwebtoken');
2
3 module.exports = function auth(req, res, next) {
4   const authHeader = req.headers.authorization || '';
5   const BEARER = 'bearer ';
6
7   if (!authHeader || !authHeader.toLowerCase().startsWith(BEARER)) {
8     return res.status(401).json({ error: 'missing_authorization_header' });
9   }
10
11   const token = authHeader.slice(BEARER.length).trim();
12   if (!token) {
13     return res.status(401).json({ error: 'missing_token' });
14   }
15
16   const secret = process.env.JWT_SECRET;
17   if (!secret) {
18     return res.status(500).json({ error: 'server_misconfigured', detail: 'JWT_SECRET is not set' });
19   }
20
21   try {
22     const payload = jwt.verify(token, secret);
23
24     const rawId = payload.userId ?? payload.id ?? payload.id_utilisateur;
25
26     const id = Number(rawId);
27     if (!Number.isFinite(id) || id <= 0) {
28       return res.status(401).json({ error: 'invalid_token_payload' });
29     }
30   }
```

FIGURE 10.24 – Middleware d'authentification JWT

Autoriser : contrôler qui a le droit d'agir

Être connecté·e ne suffit pas : je contrôle ensuite les **autorisations** sur chaque action sensible.

D'abord, je vérifie qu'un·e utilisateur.ice ne peut agir que sur ses propres réservations : si la réservation ciblée n'appartient pas à `req.user.id`, je renvoie 403 Forbidden (Figure 10.25).

```
router.patch('/:id', requireAuth, async (req, res) => {
  try {
    const id = BigInt(req.params.id);
    const { status, notes } = req.body || {};

    const current = await prisma.booking.findUnique({
      where: { id },
      include: { slot: true },
    });
    if (!current) return res.status(404).json({ error: 'not_found' });
    if (current.userId !== BigInt(req.user.id)) return res.status(403).json({ error: 'forbidden' });
  }
```

FIGURE 10.25 – Autorisation côté serveur : une personne ne peut agir que sur ses propres réservations.

Ensuite, je sécurise les actions "propriétaire" : un·e propriétaire ne peut confirmer ou annuler que des réservations liées à ses propres jardins. Je récupère le `ownerUserId` du jardin, je le compare à `req.user.id`, et si ça ne correspond pas, je renvoie 403 Forbidden. Ce contrôle protège le système même si quelqu'un appelle l'endpoint manuellement (Figure 10.26).

```
const ownerId = booking.garden?.ownerUserId;
if (!ownerId || String(ownerId) !== String(req.user.id)) {
  return res.status(403).json({ error: 'forbidden_not_owner' });
}
```

FIGURE 10.26 – Autorisation côté serveur (actions propriétaire) : contrôle strict du lien jardin → propriétaire.

Valider les entrées et renvoyer des erreurs exploitables

Enfin, j'ajoute des validations d'entrée (formats, cohérence des dates) et des réponses explicites pour que l'interface puisse afficher un message compréhensible, plutôt qu'une erreur générique. Je renvoie par exemple 400 Bad Request si les paramètres sont invalides, 404 Not Found si une ressource n'existe pas ou 409 Conflict si l'état métier empêche l'action (ex. conflit de créneau / ressource déjà prise).

Le 409 n'est pas une erreur d'authentification : c'est une erreur de **conflict métier** qui indique que la demande n'est plus compatible avec l'état actuel en base. Ces codes permettent au front d'afficher un retour clair et d'éviter les échecs silencieux (Figure 10.27).

```
if (!gardenId || !startsAt || !endsAt) {
  return res.status(400).json({ ok: false, reasons: ['invalid_params'] });
}
if (isNaN(startsAt.getTime()) || isNaN(endsAt.getTime())) {
  return res.status(400).json({ ok: false, reasons: ['invalid_dates'] });
}
if (!(startsAt < endsAt)) reasons.push('start_must_be_before_end');
if (endsAt < new Date()) reasons.push('slot_in_past');

const conflict = await hasConflictJS({ gardenId, startsAt, endsAt });
if (conflict) reasons.push('time_conflict');
```

FIGURE 10.27 – Validations et codes d'erreur : réponses explicites (400/404/409)

Éco-conception : traduction technique des choix de conception

La contrainte de **sobriété** définie au cadrage se traduit concrètement par plusieurs choix techniques dans le développement de Jardin Solidaire. L'objectif n'est pas d'optimiser à l'extrême, mais d'éviter les **chargements inutiles** et les traitements superflus.

- les listes de jardins et de réservations sont chargées de manière ciblée (pagination, filtres) plutôt que de rapatrier l'ensemble des données,
- les appels API sont **centralisés** dans des fichiers dédiés (`lib/*`) afin d'éviter les requêtes redondantes et les comportements incohérents,
- les composants sont **réutilisés** (cartes, badges de statut, actions), ce qui limite la duplication de code et simplifie la maintenance,
- les états de chargement et d'erreur sont explicités pour éviter des rafraîchissements inutiles ou des comportements ambigus côté interface.

Par exemple, la vérification de disponibilité d'un créneau passe par un endpoint dédié (`/can-book`) avec des paramètres précis (jardin, dates), afin d'éviter de charger des listes complètes ou des données non nécessaires. Cette approche permet de limiter les échanges réseau et les traitements côté serveur tout en gardant une information fiable pour l'interface (Figure 10.28).

```
frontend > src > lib > js bookings.js > Y ownerConfirmBooking
48   export async function canBook({ gardenId, startsAt, endsAt }) {
49     const params = new URLSearchParams({
50       gardenId: String(gardenId),
51       startsAt: new Date(startsAt).toISOString(),
52       endsAt: new Date(endsAt).toISOString(),
53     });
54     const res = await fetch(` ${API_BASE}/api/bookings/can-book?${params.toString()}` , {
55       headers: { ...authHeaders() },
56       cache: 'no-store',
57     });
58     return handle(res, 'check_failed');
59   }
```

FIGURE 10.28 – `canBook` : appel API ciblé (`/can-book + paramètres`) pour vérifier la disponibilité d'un créneau sans charger des données inutiles.

Ces choix contribuent à une application plus **sobre** et plus **fiable**, notamment sur des contextes mobiles ou des connexions variables, tout en conservant une expérience utilisateur.ice claire et cohérente.

Chapitre 11

CI/CD, Tests et Qualité

Pour rendre Jardin Solidaire fiable, je n'ai pas voulu dépendre uniquement de tests à la main. J'ai donc mis en place une démarche de qualité basée sur la **CI/CD** : à chaque modification, je déclenche automatiquement des vérifications, et je sais immédiatement si une évolution a cassé quelque chose.

J'ai implémenté cette automatisation avec **GitHub Actions**. Mon objectif était d'avoir un contrôle reproductible, dans un environnement neutre, qui s'exécute de la même façon à chaque *push* et à chaque *pull request*. Je définis une suite d'étapes (installation, configuration, tests), et GitHub Actions les rejoue automatiquement, sans que j'aie à tout relancer.

Dans Jardin Solidaire, j'ai créé un workflow dédié aux **tests du back-end**, parce que c'est là que se trouvent les règles sensibles (authentification, autorisations, réservation et anti-conflits). À chaque exécution, le workflow installe les dépendances, prépare un environnement de test isolé et configure une **base de données de test** séparée. J'exécute ensuite **Prisma** : je génère le client, puis j'applique les migrations sur cette base de test, pour garantir que le schéma en base correspond bien à la version du code. Une fois l'environnement prêt, les tests s'exécutent automatiquement.

All workflows			
Showing runs from all workflows			
64 workflow runs			
	Event	Status	Branch
	Actor		
✓ CI FIXED	E2E #19: Commit 73d359d pushed by thaliwoods	main	3 minutes ago ⌚ 2m 53s
✓ CI FIXED	Backend Tests #19: Commit 73d359d pushed by thaliwoods	main	3 minutes ago ⌚ 41s
✓ CI FIXED	Build Docker Images #19: Commit 73d359d pushed by thaliwoods	main	3 minutes ago ⌚ 2m 41s
✓ testing to fix ci	Build Docker Images #18: Commit 9db0ac9 pushed by thaliwoods	main	38 minutes ago ⌚ 2m 37s
✓ testing to fix ci	Backend Tests #18: Commit 9db0ac9 pushed by thaliwoods	main	38 minutes ago ⌚ 48s
✗ testing to fix ci		main	38 minutes ago ⌚ 38s

FIGURE 11.1 – Runs CI/CD (GitHub Actions)

Ce mécanisme me sert de **filet de sécurité**. Dès que je touche à une partie critique (par exemple la logique de réservation), j'obtiens un retour immédiat : si un test échoue, la pipeline passe au rouge, et je corrige tout de suite pendant que le contexte est encore frais. Ça évite d'accumuler des erreurs invisibles et ça rend l'évolution du projet plus sereine.

En complément, j'ai ajouté d'autres workflows pour couvrir la qualité du projet de manière plus globale. Par exemple, certains lancent des tests **end-to-end (E2E)** qui rejouent un parcours utilisateur.ice complet dans un navigateur, et d'autres vérifient la construction des images **Docker**. L'intérêt de Docker, c'est que je teste et je déploie exactement le même paquet : une image validée par les tests correspond à une version précise de l'application, avec ses dépendances et sa configuration.

Au final, cette approche CI/CD m'apporte deux bénéfices très concrets. D'abord, je détecte les **régressions** le plus tôt possible, avant qu'elles n'arrivent côté utilisateur.ice. Ensuite, quand tout est au vert, je peux livrer une nouvelle version avec plus de confiance, sans refaire manuellement toutes les vérifications.

```
1  name: Backend Tests
2
3  on:
4    push:
5      branches: [main]
6    pull_request:
7      branches: [main]
8
9  jobs:
10   test:
11     runs-on: ubuntu-latest
12     env:
13       NODE_ENV: test
14       DATABASE_URL: postgresql://postgres:postgres@localhost:5432/jardin_ci
15
16     services:
17       postgres:
18         image: postgres:16
19         ports: ['5432:5432']
20         env:
21           POSTGRES_USER: postgres
22           POSTGRES_PASSWORD: postgres
23           POSTGRES_DB: jardin_ci
24         options: >-
25           --health-cmd="pg_isready -U postgres"
26           --health-interval=5s
27           --health-timeout=5s
28           --health-retries=10
29
30   steps:
31     - uses: actions/checkout@v4
32
33     - uses: actions/setup-node@v4
34       with:
35         node-version: 18
36         cache: 'npm'
37         cache-dependency-path: backend/package-lock.json
38
39     - name: Install backend deps
40       working-directory: backend
41       run: npm ci
42
43     - name: Generate Prisma client
44       working-directory: backend
45       run: npx prisma generate
46
47     - name: Apply schema (deploy or push)
48       working-directory: backend
49       env:
50         DATABASE_URL: ${{ env.DATABASE_URL }}
51       run: |
52         npx prisma migrate deploy || npx prisma db push
53
```

FIGURE 11.2 – Workflow back-end : installation → base de test → Prisma → tests

Chapitre 12

Tests unitaires et tests d'intégration

Pour limiter les bugs et garder un code maintenable dans le temps, j'ai mis en place une stratégie de tests sur le back-end de Jardin Solidaire. Mon objectif est simple : quand je touche à une logique sensible (validations, sécurité, comportement des routes), je veux pouvoir relancer une suite de tests et savoir tout de suite si j'ai cassé quelque chose. Ça me permet de repérer une **régression** dès qu'elle apparaît, au lieu de la découvrir plus tard en test manuel.

Plan de tests : parcours critiques

Avant d'écrire les tests, j'ai listé les **parcours critiques** (ceux qui bloquent l'usage si ça casse) et j'ai associé à chacun un type de test, un outil et un critère de succès. Ce choix me permet de tester en priorité ce qui a le plus d'impact utilisateur.ice, de rendre la stratégie lisible et ne pas tester pas au hasard.

Ce plan de tests couvre l'ensemble de la stratégie (unitaires, intégration et E2E). Les scénarios E2E listés ici sont détaillés dans le chapitre "Tests end-to-end (E2E)", où je présente les parcours rejoués côté interface et leur exécution en CI.

Parcours critique	Type	Outil	Critère de succès
Inscription / connexion	E2E	Playwright	Page accessible, champs présents, soumission possible
Réserver un créneau	Intégration	Supertest	201 et réservation créée
Conflit de réservation	Intégration	Supertest	409 Conflict si créneau déjà pris
Accès non autorisé	Intégration	Supertest	401 si non authentifié·e / 403 si non autorisé·e
Navigation jardins → détail	E2E	Playwright	Au moins une carte visible, clic sur une carte, page détail affichée

FIGURE 12.1 – Plan de tests : association des parcours critiques, types de tests, outils et critères de succès.

J'ai mis en place le projet avec des commandes dédiées pour exécuter les tests à tout moment : `npm run test` (suite complète), `npm run test:watch` (feedback immédiat en développement) et `npm run test:cov` (couverture) (Figure 12.2). Ce choix me permet de vérifier rapidement l'impact d'un changement et de détecter une régression avant même d'ouvrir l'interface.

```
"scripts": {  
  "test": "jest",  
  "test:watch": "jest --watch",  
  "test:cov": "jest --coverage",  
  "start": "node server.js",  
  "dev": "node server.js",  
  "prisma:generate": "prisma generate --schema=./prisma/schema.prisma",  
  "prisma:migrate": "prisma migrate dev --schema=./prisma/schema.prisma",  
  "prisma:studio": "prisma studio --schema=./prisma/schema.prisma",  
  "prisma:seed": "node prisma/seed.js",  
  "seed": "prisma db seed",  
  "reset:seed": "prisma migrate reset --force"  
},
```

FIGURE 12.2 – Scripts de test back-end

J'ai organisé mes tests en deux niveaux complémentaires : **tests unitaires** et **tests d'intégration**. Avec cette approche, je vérifie à la fois la logique "en détail" (fonction par fonction) et le comportement réel de l'API lorsqu'elle est appelée comme en production (routes Express, statuts, réponses).

Tests unitaires

J'utilise les tests unitaires pour sécuriser des fonctions isolées : validations, logique métier simple, et fonctions critiques comme le hash de mot de passe. Ici, je teste **un comportement précis** sans dépendre du reste : je ne démarre ni serveur ni base de données. Je donne une entrée à une fonction et je vérifie que la sortie est exactement celle attendue.

Par exemple, je teste une fonction de validation qui liste les champs obligatoires manquants. J'ai choisi de couvrir plusieurs cas limites (valeurs vides, espaces, champs manquants) parce que ce sont eux qui provoquent le plus souvent des bugs en production (Figure 12.3).

```
const { getMissingRequiredFields } = require('../utils/formUtils');

describe('getMissingRequiredFields (backend)', () => {
  test('returns [] when name and description are properly filled', () => {
    const formData = {
      name: 'Alice Dupont',
      description: "J'ai 5 ans d'expérience en jardinage",
      location: 'Paris',
      area: '20m²',
      services: 'Potager',
      photos: []
    };
    expect(getMissingRequiredFields(formData)).toEqual([]);
  });

  test('flags "name" if the name field is empty or whitespace', () => {
    const formData = {
      name: ' ',
      description: 'Il fait beau dans le jardin',
      location: '',
      area: '',
      services: '',
      photos: []
    };
    expect(getMissingRequiredFields(formData)).toEqual(['name']);
  });

  test('flags "description" if the description field is empty', () => {
    const formData = {
      name: 'Bob Martin',
      description: '',
      location: 'Lyon',
      area: '',
      services: '',
      photos: []
    };
    expect(getMissingRequiredFields(formData)).toEqual(['description']);
  });

  test('flags ["name","description"] if both fields are empty', () => {
    const formData = {
      name: '',
      description: '',
      location: 'Marseille',
      area: '',
      services: '',
      photos: []
    };

    const missing = getMissingRequiredFields(formData);
    expect(missing.sort()).toEqual(['description', 'name']);
  });

  test('ignores other fields (only name & description are required)', () => {
    const formData = {
      name: 'Carole',
      description: 'Je jardine en ville',
      location: '',
      area: '',
      services: '',
      photos: []
    };
    expect(getMissingRequiredFields(formData)).toEqual([]);
  });
});
```

FIGURE 12.3 – Test unitaire : `getMissingRequiredFields` identifie les champs obligatoires manquants ou vides.

Je sécurise aussi une fonction critique côté sécurité : le hash de mot de passe. Le test vérifie que le mot de passe n'est jamais stocké en clair et que bcrypt valide bien le hash généré, tout en rejetant un mauvais mot de passe (Figure 12.4).

```
1  const bcrypt = require('bcrypt');
2  const { hashPassword } = require('../utils');
3
4  describe('Password hashing', () => {
5      it('hashes a password and verifies it with bcrypt', async () => {
6          const password = 'Test123!';
7          const hashed = await hashPassword(password);
8
9          expect(hashed).toBeDefined();
10         expect(typeof hashed).toBe('string');
11         expect(hashed).not.toBe(password);
12
13         const ok = await bcrypt.compare(password, hashed);
14         expect(ok).toBe(true);
15
16         const bad = await bcrypt.compare('wrong', hashed);
17         expect(bad).toBe(false);
18     });
19 })
```

FIGURE 12.4 – Test unitaire sécurité : hashPassword produit un hash non réversible et validable par bcrypt.compare, tout en refusant un mauvais mot de passe.

Ces tests sont **rapides** et me donnent un retour immédiat pendant que je développe.

Tests d'intégration

En complément, j'ai ajouté des tests d'intégration sur l'API avec **Supertest**. Je teste l'application **comme elle est réellement utilisée** : j'envoie une requête HTTP sur une route Express, je vérifie le statut, puis je contrôle le format et le contenu de la réponse.

Un exemple simple mais essentiel : un smoke test qui vérifie que le serveur répond bien et que les routes inconnues renvoient une erreur JSON claire. Ce type de test me donne un signal immédiat si l'API est opérationnelle et correctement configurée (Figure 12.5).

```
1  const request = require('supertest');
2  const app = require('../server');
3
4  describe('Backend smoke tests', () => {
5    it('GET / returns 200 and a ping text', async () => {
6      const res = await request(app).get('/');
7      expect(res.status).toBe(200);
8      expect(typeof res.text).toBe('string');
9      expect(res.text).toMatch(/Prisma backend is online/i);
10   });
11
12  it('Unknown route returns 404 JSON', async () => {
13    const res = await request(app).get('/does-not-exist');
14    expect(res.status).toBe(404);
15    expect(res.body).toHaveProperty('error', 'NOT_FOUND');
16  });
17});
```

FIGURE 12.5 – Test d'intégration (smoke)

Je teste aussi une route de lecture réaliste, GET /api/gardens, en vérifiant non seulement le 200, mais aussi la structure des données renvoyées (clés attendues, types, présence des coordonnées). J'ai choisi ce niveau de vérification parce que c'est exactement ce dont le front dépend : une réponse stable, cohérente, et exploitable (Figure 12.6).

```
node > tests > JS gardens.test.js > ⚡ describe('GET /api/gardens/:id') callback > ⚡ it('returns 404 for non-existing id', () => {
  const request = require('supertest');
  const app = require('../server');

  describe('GET /api/gardens', () => {
    it('returns 200 and an array of gardens with expected keys', async () => {
      const res = await request(app).get('/api/gardens');
      expect(res.status).toBe(200);
      expect(Array.isArray(res.body)).toBe(true);

      if (res.body.length > 0) {
        const g = res.body[0];

        expect(g).toEqual(expect.objectContaining({
          id: expect.any(String),
          title: expect.any(String),
          status: expect.any(String),
        }));

        expect(g).toHaveProperty('lat');
        expect(g).toHaveProperty('lng');
        expect(typeof g.lat).toBe('number');
        expect(typeof g.lng).toBe('number');

        expect(g).toHaveProperty('address');
        expect(typeof g.address).toBe('string');

        expect(g).toHaveProperty('photos');
        expect(Array.isArray(g.photos)).toBe(true);

        if (g.averageRating != null) expect(typeof g.averageRating).toBe('number');
        if (g.publishedAt != null) expect(typeof g.publishedAt).toBe('string');
      }
    });
  });
});
```

FIGURE 12.6 – Test d'intégration

À ce stade du projet, j'ai choisi de concentrer mes tests d'intégration sur des routes de lecture (GET). Elles sont plus simples à rendre reproductibles, car elles ne modifient pas l'état de la base et limitent les effets de bord.

Les routes d'écriture (POST, PATCH, DELETE) demandent une stratégie plus stricte : base de test dédiée et nettoyage systématique entre chaque test, pour repartir d'un état connu et éviter que les tests s'influencent entre eux. J'ai donc priorisé des scénarios lisibles et fiables à relancer, tout en gardant la base nécessaire pour étendre la stratégie sur les routes d'écriture plus tard.

Au final, cette combinaison me donne un cadre fiable pour faire évoluer Jardin Solidaire. Les tests unitaires verrouillent la logique, et les tests d'intégration confirment que l'API répond correctement dans des scénarios proches de l'usage réel.

Chapitre 13

Tests end-to-end (E2E)

En complément des tests unitaires et d'intégration, j'ai mis en place des tests **end-to-end (E2E)** avec **Playwright**. Mon objectif est de vérifier que Jardin Solidaire fonctionne **comme un.e utilisateur.ice le vit**, à l'écran, et pas seulement en théorie via une fonction ou une réponse JSON.

Avec les E2E, je rejoue des scénarios complets du début à la fin : un navigateur ouvre l'interface Next.js, l'utilisateur.ice clique, navigue, remplit un formulaire, et l'application déclenche des appels à l'API Express. Ce type de test traverse les couches (front, API, et parfois base) et me permet de valider le parcours réel : est-ce que la page charge, est-ce que la navigation fonctionne, est-ce que les éléments attendus s'affichent, est-ce que je finis au bon endroit ?

J'ai ciblé en priorité les parcours critiques : charger une page essentielle, naviguer de la liste des jardins vers une fiche jardin, vérifier qu'un formulaire clé est bien rendu, et contrôler que certains comportements changent correctement selon qu'on est connecté·e ou non (par exemple les favoris).

Un exemple représentatif est le scénario `gardens.spec.js`. Dans ce test, je simule un usage réel : j'ouvre la page des jardins, je vérifie qu'au moins une carte s'affiche, puis je clique sur un jardin et je contrôle que j'arrive bien sur la page de détail (Figure 13.1). J'ai choisi ce scénario parce qu'il sécurise un point de friction classique : si la liste ne rend rien, ou si la navigation casse, l'utilisateur.ice ne peut tout simplement pas avancer. Ici, je valide donc l'expérience complète : chargement, navigation, rendu, et cohérence du parcours.

```
1  const { test, expect } = require('@playwright/test');
2
3  test('gardens list shows at least one garden, and detail loads', async ({ page }) => {
4    await page.goto('/gardens', { waitUntil: 'domcontentloaded' });
5
6    const jardinsTab = page.getByRole('tab', { name: /jardins/i });
7    if (await jardinsTab.count() < 1) {
8      await jardinsTab.first().click();
9    }
10
11   const cards = page.locator(' [data-testid="garden-card"]', { href: '/gardens/' });
12
13   await cards.first().waitFor({ state: 'attached', timeout: 10_000 });
14
15   const first = cards.first();
16   await first.scrollIntoViewIf Needed().catch(() => {});
17
18   const href = await first.getAttribute('href').catch(() => null);
19   if (href) {
20     await page.goto(href);
21   } else {
22     await first.click({ trial: true }).catch(() => {});
23     await first.click();
24   }
25
26   const heading = page.getByRole('heading').first();
27   await expect(heading).toBeVisible({ timeout: 10_000 });
28 });


```

FIGURE 13.1 – `gardens.spec.js`

J'ai aussi écrit un scénario `register.spec.js` pour valider qu'un parcours clé "formulaire" reste utilisable : présence d'un titre, champ email, champ mot de passe et bouton de soumission. J'ai fait ce choix parce que ces éléments sont des prérequis UX : si un champ disparaît, si un label change, ou si la page ne rend rien, l'utilisateur.ice est bloqué.e immédiatement. Pour rendre le test robuste, j'ai ajouté des fallbacks (par label, type d'input, placeholder), afin qu'un changement mineur de wording ne casse pas le test, tout en détectant les vraies régressions (Figure 13.2).

```

const heading = page.getByRole('heading', {
  name: /inscription|register|créer un compte|créez votre compte|sign up/i,
});

const headingVisible = await heading.isVisible().catch(() => false);
if (!headingVisible) {
  await expect(page.getByRole('heading').first()).toBeVisible();
} else {
  await expect(heading).toBeVisible();
}

const emailByLabel = page.getByLabel(/email|e-mail|courriel/i);
const hasEmailLabel = await emailByLabel.count();
if (hasEmailLabel) {
  await expect(emailByLabel.first()).toBeVisible();
} else {
  const emailByType = page.locator('input[type="email"]');
  const emailByPlaceholder = page.getByPlaceholder(/email|e-mail|courriel/i);
  const anyEmail =
    (await emailByType.count()) ||
    (await emailByPlaceholder.count());
  expect(anyEmail, 'Email input not found').toBeGreaterThan(0);
}

const pwdByLabel = page.getByLabel(/mot de passe|password/i);
const hasPwdLabel = await pwdByLabel.count();
if (hasPwdLabel) {
  await expect(pwdByLabel.first()).toBeVisible();
} else {
  const pwdByType = page.locator('input[type="password"]');
  const pwdByPlaceholder = page.getByPlaceholder(/mot de passe|password/i);
  const anyPwd =
    (await pwdByType.count()) ||
    (await pwdByPlaceholder.count());
  expect(anyPwd, 'Password input not found').toBeGreaterThan(0);
}

const submit = page.getByRole('button', {
  name: /inscription|s'inscrire|créer|create|register|sign up/i,
});
const hasSubmit = await submit.count();
if (hasSubmit) {
  await expect(submit.first()).toBeVisible();
} else {
  const anySubmit = page.locator('button[type="submit"]');
  await expect(anySubmit.first()).toBeVisible();
}

console.log(`✅ Registration page checked at: ${used}`);
}

```

FIGURE 13.2 – `register.spec.js`

Pour écrire des tests plus lisibles, je garde une structure inspirée de *Given / When / Then*. Je pars d'un état de départ, je déclenche une action, puis je vérifie le résultat attendu. Cette logique m'aide à éviter les scénarios confus et à produire des tests faciles à relire, donc plus simples à maintenir.

Enfin, j'ai intégré ces tests E2E à la **CI/CD** via **GitHub Actions**. À chaque push ou Pull Request, je lance automatiquement un environnement de test reproductible : un service Postgres, l'application du schéma Prisma, un seed de données, puis l'exécution Playwright avec génération d'un rapport HTML. J'ai choisi de l'automatiser dans la CI pour détecter une régression le plus tôt possible : si un changement casse un parcours important, je le vois immédiatement, avant déploiement, et je peux corriger rapidement, avec une compréhension encore claire du changement introduit (Figure 13.3).

```
o > workflows > e2e.yml
  name: E2E

  on:
    push:
      branches: [ main ]
    pull_request:
      branches: [ main ]

  jobs:
    e2e:
      runs-on: ubuntu-latest

      services:
        postgres:
          image: postgres:16
          ports: ['5432:5432']
          env:
            POSTGRES_USER: postgres
            POSTGRES_PASSWORD: postgres
            POSTGRES_DB: jardin_e2e
          options: >-
            --health-cmd="pg_isready -U postgres"
            --health-interval=5s
            --health-timeout=5s
            --health-retries=10

      env:
        DATABASE_URL: postgresql://postgres:postgres@localhost:5432/jardin_e2e
        NODE_ENV: test
        NEXT_PUBLIC_API_URL: http://127.0.0.1:5001
        E2E_AUTH: "0"
        PORT_FRONT: "3000"
        PORT_BACK: "5001"
        CI: "1"
```

FIGURE 13.3 – Extrait CI E2E

Au final, les tests unitaires et d'intégration me protègent sur la logique et le comportement de l'API, tandis que les E2E me protègent sur les parcours réels. Ensemble, ils rendent Jardin Solidaire plus fiable et réduisent les mauvaises surprises au moment du déploiement.

Chapitre 14

Déploiement

Déploiement - schéma d'architecture

Pour mettre Jardin Solidaire en ligne, j'ai choisi un déploiement **simple et proche d'un setup professionnel**, avec une séparation claire des rôles : le front **Next.js** sur **Vercel**, l'**API Node/Express** sur **Render**, et la base **PostgreSQL** sur **Neon**. Mon objectif était d'avoir un MVP **opérable**, facile à maintenir, et facile à faire évoluer.

Le fonctionnement est le suivant : l'utilisateur.ice passe toujours par le front. C'est lui qui affiche les pages, les formulaires et l'interface. Dès qu'une action a besoin de données (lister des jardins, créer une réservation, charger un profil, envoyer un message), j'envoie une requête HTTP depuis le front vers l'API. Ensuite, l'API applique les règles métier (droits, validations, anti-conflits) et lit/écrit en base sur Neon via Prisma, puis renvoie une réponse au front. Ce flux front → API → base → front me permet de garder des responsabilités nettes : **le front affiche, l'API décide, la base stocke** (Figure 14.1).

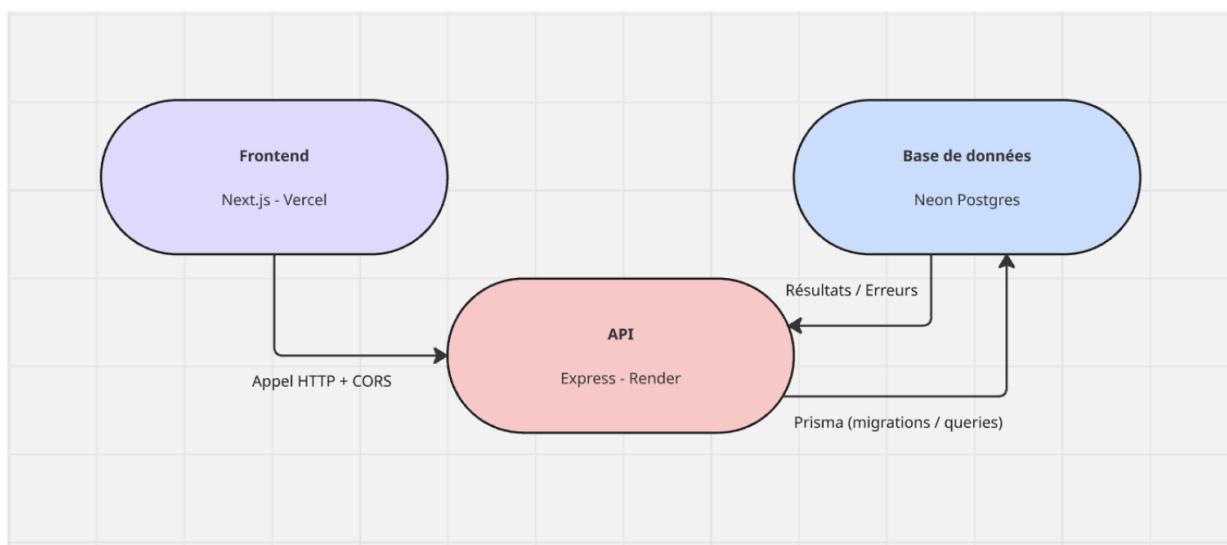


FIGURE 14.1 – Schéma d'architecture de déploiement (Vercel / Render / Neon).

Côté sécurité, j'ai configuré le **CORS** pour que mon API n'accepte que les requêtes provenant du domaine du front. Concrètement, j'ai externalisé cette configuration côté Render via des variables d'environnement (Figure 14.3) : je garde les secrets hors du code (JWT, URLs), je limite les origines autorisées, et je peux ajuster la configuration sans modifier le repository.

Déploiement - plateformes et choix techniques

J'ai documenté précisément le rôle de chaque plateforme, parce que je voulais un déploiement où chaque brique reste remplaçable sans casser les autres : le front reste un client HTTP, l'API centralise la logique métier, et la base conserve l'état.

Sur **Vercel**, j'ai déployé le front Next.js : le dépôt est connecté et chaque push sur la branche principale déclenche automatiquement un build puis un déploiement. J'ai choisi ce fonctionnement pour réduire la friction : je push, je vérifie en ligne, et je garde une trace claire des versions déployées (Figure 14.2).

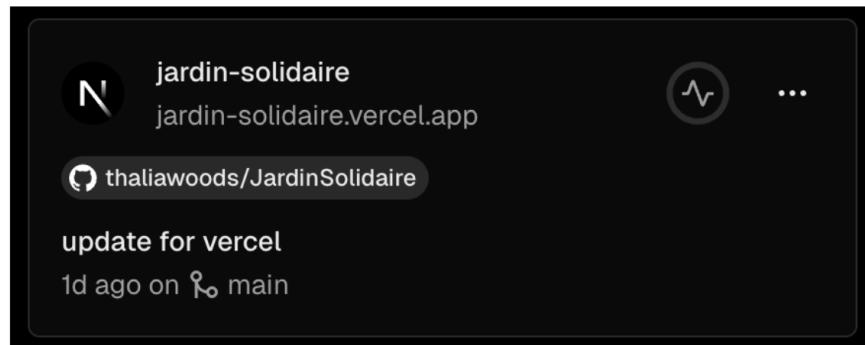


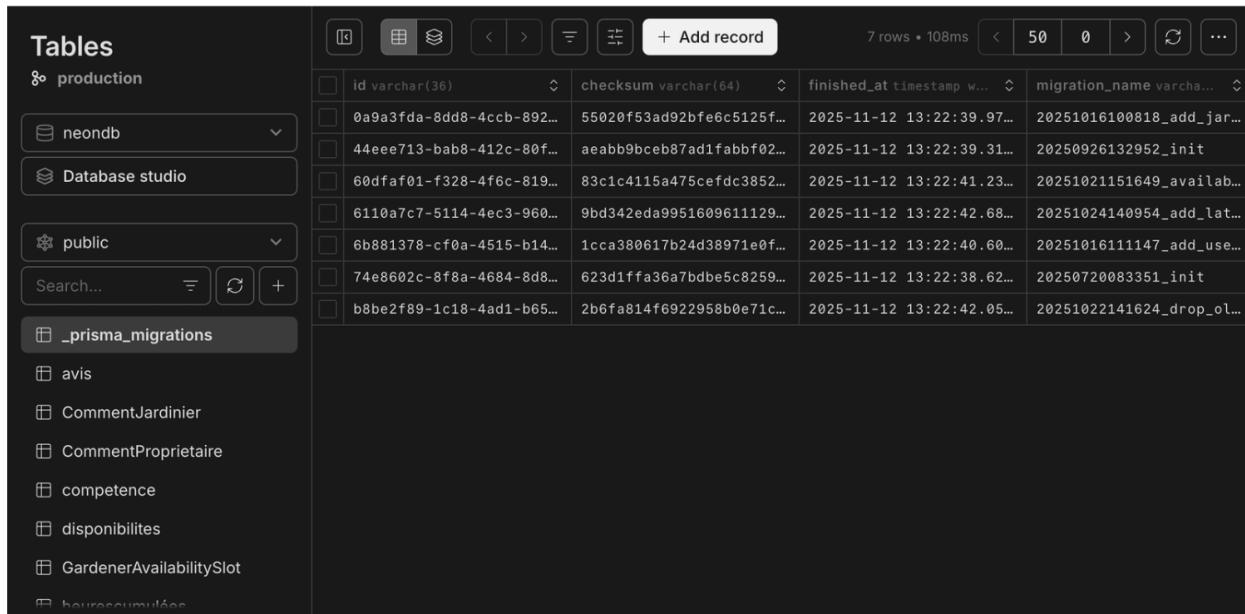
FIGURE 14.2 – Vercel : déploiement automatique relié à GitHub (chaque push sur `main` déclenche un build + un déploiement).

Sur **Render**, j'ai déployé l'API Node/Express depuis le dépôt GitHub. J'ai fait le choix de piloter la configuration uniquement via des variables d'environnement : c'est la configuration qui change selon l'environnement, pas le code. Ça me permet aussi de séparer proprement local et production, tout en évitant de versionner des secrets.

KEY	VALUE
DATABASE_URL
DIRECT_URL
FRONTEND_ORIGIN
JWT_SECRET
NODE_ENV

FIGURE 14.3 – Render : configuration par variables d'environnement pour garder les secrets hors du code et piloter la sécurité (CORS / JWT).

Sur **Neon**, j'ai choisi une base PostgreSQL managée pour ne pas perdre de temps sur l'infrastructure. Mon objectif était de me concentrer sur le produit, tout en gardant une base centralisée et cohérente. Neon s'intègre bien avec Prisma : je garde un historique clair de l'évolution du schéma grâce aux migrations, ce qui rend le setup reproductible et auditable (Figure 14.4).



The screenshot shows the Neon PostgreSQL interface. On the left, a sidebar lists databases: 'neondb' (selected), 'Database studio', 'public', and a search bar. Below these are tables: '_prisma_migrations', 'avis', 'CommentJardinier', 'CommentProprietaire', 'competence', 'disponibilites', 'GardenerAvailabilitySlot', and 'heurescumulées'. The main area displays the '_prisma_migrations' table with the following data:

	<code>id</code> varchar(36)	<code>checksum</code> varchar(64)	<code>finished_at</code> timestamp w...	<code>migration_name</code> varchar...
1	0a9a3fda-8dd8-4ccb-892...	55020f53ad92bfe6c5125f...	2025-11-12 13:22:39.97...	20251016100818_add_jar...
2	44eee713-bab8-412c-80f...	aebabb9cceb87ad1fabbf02...	2025-11-12 13:22:39.31...	20250926132952_init
3	60dfaf01-f328-4f6c-819...	83c1c4115a475cefdc3852...	2025-11-12 13:22:41.23...	20251021151649_availab...
4	6110a7c7-5114-4ec3-960...	9bd342eda9951609611129...	2025-11-12 13:22:42.68...	20251024140954_add_lat...
5	6b881378-cf0a-4515-b14...	1cca380617b24d38971e0f...	2025-11-12 13:22:40.60...	20251016111147_add_use...
6	74e8602c-8f8a-4684-8d8...	623d1ffa36a7bdb5c8259...	2025-11-12 13:22:38.62...	20250720083351_init
7	b8be2f89-1c18-4ad1-b65...	2b6fa814f6922958b0e71c...	2025-11-12 13:22:42.05...	20251022141624_drop_ol...

FIGURE 14.4 – Neon (PostgreSQL) : suivi des migrations Prisma en production (`_prisma_migrations`) pour versionner l'évolution du schéma et garder un historique fiable.

Au final, ce trio **Vercel / Render / Neon** me donne un déploiement simple à opérer, réaliste en contexte professionnel, et solide pour un MVP : responsabilités séparées, secrets hors du code, et une base versionnée proprement via Prisma.

Chapitre 15

Limites, améliorations et perspectives

Aujourd’hui, Jardin Solidaire est un MVP **testable** : on peut créer un compte, publier ou consulter des jardins, proposer des créneaux, réserver et échanger. Ce socle me permet déjà de confronter le produit à un usage réel. Cependant il a encore des limites : il fonctionne, sans encore être optimisé pour un usage en conditions réelles.

Le premier enjeu, est de passer d’un simple catalogue à une recherche vraiment **pertinente**. Pour l’instant, on peut trouver des jardins, mais pas forcément **celui qui nous convient** rapidement. Or, dans la réalité, tout se joue sur des critères très concrets : la distance, les disponibilités, le besoin précis, et les conditions sur place. La priorité est donc de rendre la recherche plus efficace avec de la **géolocalisation** et des **filtres utiles** (durée estimée, outils disponibles, point d’eau, accessibilité, niveau d’autonomie). L’objectif est de réduire le temps passé à ouvrir des fiches pour finalement se rendre compte que ce n’est pas adapté.

Une fois qu’on trouve un jardin pertinent, il faut que l’expérience soit **rassurante**. Techniquement, l’application est sécurisée (authentification, autorisations, validations), mais Jardin Solidaire touche à un contexte sensible : on accueille quelqu’un chez soi, ou on se rend chez un inconnu. Pour que l’usage soit réellement fluide, je dois ajouter des mécanismes **côté usage** : avis, signalement et modération, afin de poser un cadre clair, de confiance, et éviter les zones grises qui freinent les utilisateur.ices.

Ensuite, je veux que cette expérience reste **accessible** au plus grand nombre. J’ai déjà posé des bases (labels, focus visible, messages clairs), mais je veux aller plus loin : renforcer la navigation clavier, vérifier les contrastes et éviter que des interactions trop visuelles (carte, hover) bloquent une partie des utilisateur.ices. Jardin Solidaire vise l’inclusion donc l’interface doit rester simple et utilisable pour des personnes très différentes.

Enfin, je pourrai ajouter ce qui fait une partie de l’identité du projet : la **banque du temps**. Je la vois comme une manière de valoriser l’engagement sans monétiser le service, mais je veux la concevoir avec prudence, pour éviter l’effet score ou les abus, et rester dans une logique d’encouragement plutôt que de performance.

Les prochaines étapes sont donc claires : **faciliter la trouvaille, mieux rassurer, mieux inclure**, puis introduire la banque du temps sans dénaturer l’entraide.

Chapitre 16

Conclusion

Jardin Solidaire répond à un besoin très concret : faire revivre des jardins sous-utilisés en les mettant en lien avec des personnes qui ont envie de jardiner, d'apprendre et de donner un coup de main. Derrière cette idée simple, il y a des enjeux importants : recréer du lien de proximité, faciliter la transmission de savoir-faire, et redonner une place au vivant, même en ville. L'objectif n'est pas de créer un service marchand, mais un outil d'entraide, où l'échange repose surtout sur le temps, la coopération et la confiance.

Ce projet a été une vraie montée en compétences, parce qu'il m'a obligée à aller au bout des choses, comme dans un contexte pro. J'ai appris à partir d'un besoin réel, à le clarifier, puis à le traduire en parcours utilisateur.ices et en fonctionnalités cohérentes. J'ai aussi dû faire des choix : définir un périmètre atteignable, prioriser, et accepter que tout ne pouvait pas être fait de suite, tant que le socle n'était pas solide.

Un point central a été de construire une logique métier fiable autour de la réservation : qui peut réserver, dans quelles conditions, comment éviter les conflits de créneaux, et comment suivre l'évolution d'une demande (en attente, confirmée, annulée, etc). C'est ce type de règles qui fait la différence entre une démo et une application qui tient dans le temps. J'ai dû bien modéliser les données et structurer la base PostgreSQL via Prisma, afin d'éviter les incohérences et de préparer les évolutions futures.

Sur le plan technique, j'ai mis en place une architecture full-stack complète. Cela m'a permis de travailler comme sur un vrai produit : organiser le code, séparer les responsabilités, gérer les erreurs proprement, et construire une API claire côté serveur.

J'ai aussi beaucoup travaillé la sécurité, avec une approche contrôle systématique côté serveur : ne jamais supposer qu'une requête est légitime juste parce que l'utilisateur.ice est connecté. Concrètement, ça veut dire authentifier, vérifier les autorisations, valider les données reçues, et renvoyer des erreurs explicites quand une action n'est pas permise.

Enfin, j'ai progressé sur la qualité et la fiabilité : tests unitaires, tests d'intégration et end-to-end, puis CI/CD et déploiement. L'intégration continue m'a appris à travailler avec des filets de sécurité, et le déploiement m'a forcée à rendre l'application reproductible et stable. Au final, ça m'a aidée à limiter les régressions et à livrer plus sereinement.

Aujourd'hui, Jardin Solidaire est un MVP fonctionnel : une première version utilisable qui couvre le cœur du produit. Cette base est suffisamment solide pour commencer à tester le service en conditions réelles, avec des habitants, des associations ou des initiatives locales. L'idée est de confronter le produit à de vrais usages, récolter des retours, et améliorer à partir de situations concrètes.

Pour finir, ce projet est pour moi une démonstration de ma capacité à mener une application de bout en bout : comprendre un besoin, concevoir des parcours, structurer une architecture, poser des règles métier, sécuriser, tester, déployer et itérer. Et il ouvre naturellement sur la suite : continuer à construire des outils utiles, simples à utiliser, et pensés pour les gens qui vont vraiment s'en servir.

Annexe A — Questionnaire complet

Questionnaire (Google Form) : <https://docs.google.com/forms/d/10hzCtKUQK7Z44ke8tMdHkosV9gpD1hxhAgjMddECBm8/edit>

The screenshot shows a Google Form interface. On the left, there's a sidebar with the form title 'Formulaire JardinSolidaire' and sections for 'Introduction' and 'Le concept de JardinSolidaire :'. The main area contains two questions: one about garden maintenance frequency and another about tasks needing help. Both questions have a 'Autre:' input field at the end.

Section : Propriétaires de jardins
(À remplir uniquement si vous avez un jardin.)

À quelle fréquence entretez-vous votre jardin ?

Une fois par semaine
 Une fois par mois
 Occasionnellement, seulement en cas de besoin

Quelles tâches nécessitent le plus d'aide dans votre jardin ?

Tonte de pelouse
 Désherbage
 Taille des haies
 Plantation de fleurs ou légumes
 Arrosage
 Autre : _____

Extrait du formulaire Google Forms

Formulaire JardinSolidaire

Introduction

Nous travaillons sur un projet visant à connecter les propriétaires de jardins avec des **amis du vert** amateurs ou passionnés, dans un système collaboratif basé sur l'échange de temps. Ce questionnaire a pour but de mieux comprendre vos besoins, attentes et préférences, afin de concevoir une plateforme simple, utile et adaptée.

Le concept de JardinSolidaire :

- **Pour les propriétaires de jardins** : Publier une annonce pour décrire leur jardin et leurs besoins (désherbage, tonte, plantation, etc.), planifier des créneaux via un calendrier intégré, et sélectionner des **amis du vert** selon leurs compétences et disponibilités.
- **Pour les amis du vert** : Trouver des jardins proches grâce à la géolocalisation et des filtres précis, réserver des créneaux pour participer à l'entretien d'un jardin, et cumuler des heures d'échange à utiliser pour accéder à d'autres jardins.

Vos réponses nous permettront de mieux comprendre comment concevoir une plateforme répondant à vos attentes et d'identifier les fonctionnalités clés. Merci pour votre participation !

* Indique une question obligatoire

Section : Propriétaires de jardins

(À remplir uniquement si vous avez un jardin.)

1. À quelle fréquence entretez-vous votre jardin ?

Plusieurs réponses possibles.

- Une fois par semaine
- Une fois par mois
- Occasionnellement, seulement en cas de besoin

2. Quelles tâches nécessitent le plus d'aide dans votre jardin ?

Plusieurs réponses possibles.

- Tonte de pelouse
- Désherbage
- Taille des haies
- Plantation de fleurs ou légumes
- Arrosage
- Autre : _____

3. Seriez-vous prêt(e) à accueillir des amis du vert pour vous aider ?

Plusieurs réponses possibles.

- Oui
- Non
- Peut-être

4. Quels critères seraient importants pour choisir un ami du vert ?

Plusieurs réponses possibles.

- Niveau d'expérience
- Avis et recommandations
- Outils disponibles
- Disponibilité immédiate
- Autre : _____

Section : Amis du vert

(À remplir uniquement si vous n'avez pas de jardin ou si vous souhaitez jardiner ailleurs.)

5. Pourquoi aimeriez-vous utiliser une plateforme comme JardinSolidaire ?

Plusieurs réponses possibles.

- Apprendre le jardinage
- Accéder à un espace vert pour se détendre
- Aider des propriétaires de jardins
- Cumuler des heures pour accéder à des jardins privés
- Autre : _____

6. Quelles compétences avez-vous déjà en jardinage ?

Plusieurs réponses possibles.

- Aucune, je suis débutant(e).
- Quelques bases (tonte, désherbage, etc.).
- Avancé(e), je peux gérer des tâches complexes.

7. Quelles tâches préférez-vous réaliser en tant qu'ami du vert ?

Plusieurs réponses possibles.

- Plantation
- Arrosage
- Désherbage
- Tonte
- Taille des haies
- Autre : _____

8. À quelle fréquence seriez-vous disponible pour jardiner ?

Plusieurs réponses possibles.

- Une fois par semaine
- Une fois par mois
- Occasionnellement, seulement en cas de besoin

9. Seriez-vous prêt(e) à vous déplacer en dehors de votre zone de résidence pour jardiner ?

Plusieurs réponses possibles.

- Oui
- Non
- Cela dépend de la distance.

Section : Fonctionnalités et attentes

10. Quelles sont vos principales préoccupations concernant cette plateforme ?

Plusieurs réponses possibles.

- Sécurité des échanges (fiabilité des utilisateurs)
- Confidentialité des données personnelles
- Qualité des interventions (compétences des amis du vert)
- Difficulté d'utilisation de la plateforme
- Autre : _____

11. Quels aspects pourraient vous motiver à utiliser cette application régulièrement ?

Plusieurs réponses possibles.

- Facilité d'utilisation
- Communauté active et fiable
- Large choix de jardins ou d'amis du vert
- Système d'échange équitable
- Autre : _____

12. Pensez-vous que ce projet est pertinent et envisagez-vous d'utiliser ce site ?

Plusieurs réponses possibles.

- Oui
- Non
- Autre : _____

13. Avez-vous des suggestions ou des idées pour améliorer ce projet ?

Section : Profil général

14. Quel est votre âge ? *

Plusieurs réponses possibles.

- Moins de 18 ans
- 18-30 ans
- 31-45 ans
- 46-60 ans
- Plus de 60 ans

15. Où habitez-vous ? *

Plusieurs réponses possibles.

- Zone urbaine
- Zone périurbaine
- Zone rurale

16. Possédez-vous un jardin ? *

Plusieurs réponses possibles.

- Oui
- Non

17. Si oui, quelle est sa taille approximative ?

Plusieurs réponses possibles.

- Petite (< 50 m²)
- Moyenne (50-100 m²)
- Grande (> 100 m²)

18. Quelle est votre relation actuelle avec le jardinage ? *

Plusieurs réponses possibles.

- J'entretiens régulièrement mon jardin.
 - J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
 - Je n'ai pas de jardin, mais j'aimerais en entretenir un.
 - Le jardinage ne m'intéresse pas.
-

Ce contenu n'est ni rédigé, ni cautionné par Google.

Google Forms

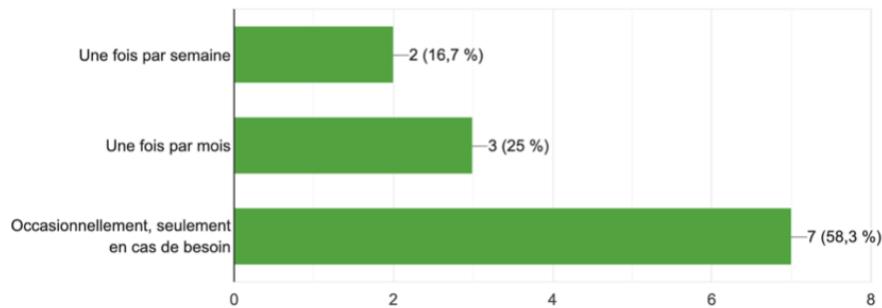
Annexe B — Résultats détaillés du questionnaire

Résultats (exports Google Forms) : graphiques et export complet.

Section : Propriétaires de jardins

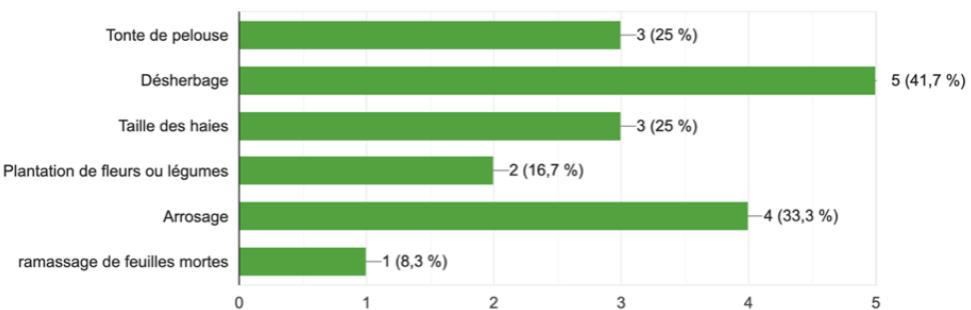
À quelle fréquence entretenez-vous votre jardin ?

12 réponses



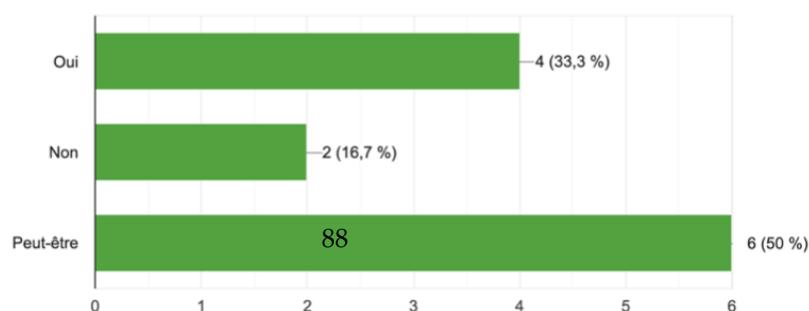
Quelles tâches nécessitent le plus d'aide dans votre jardin ?

12 réponses

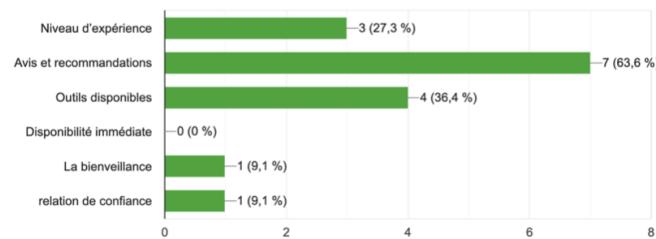


Seriez-vous prêt(e) à accueillir des amis du vert pour vous aider ?

12 réponses

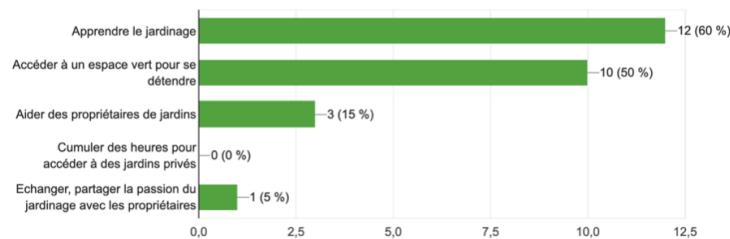


Quels critères seraient importants pour choisir un ami du vert ?
11 réponses

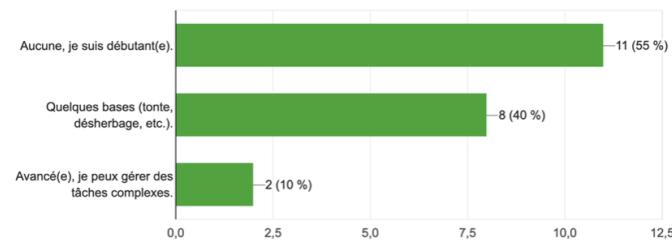


Section : Amis du vert

Pourquoi aimeriez-vous utiliser une plateforme comme JardinSolidaire ?
20 réponses

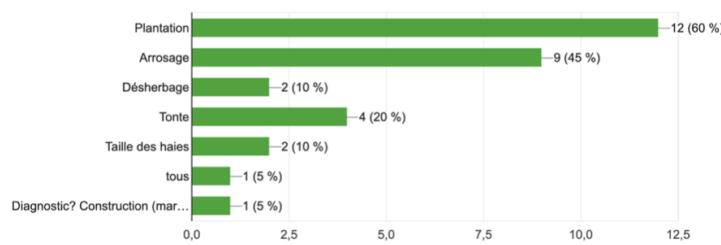


Quelles compétences avez-vous déjà en jardinage ?
20 réponses



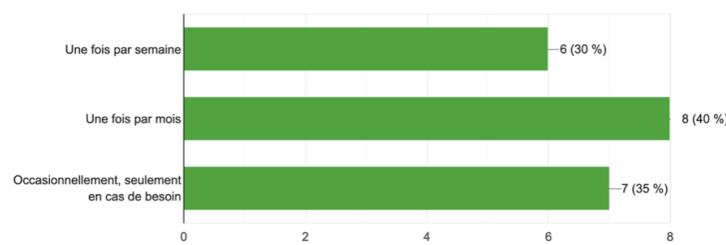
Quelles tâches préférez-vous réaliser en tant qu'amis du vert ?

20 réponses



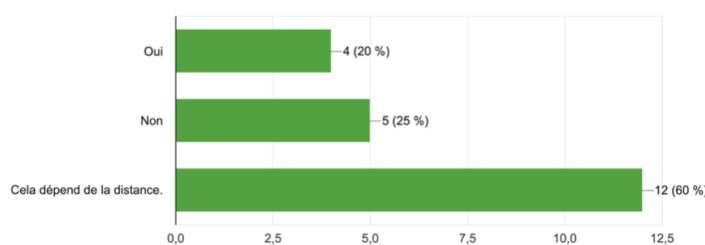
À quelle fréquence seriez-vous disponible pour jardiner ?

20 réponses



Seriez-vous prêt(e) à vous déplacer en dehors de votre zone de résidence pour jardiner ?

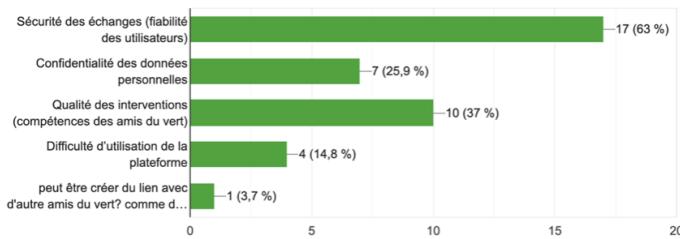
20 réponses



Section : Fonctionnalités et attentes

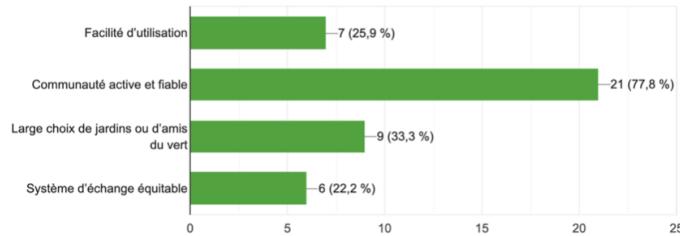
Quelles sont vos principales préoccupations concernant cette plateforme ?

27 réponses



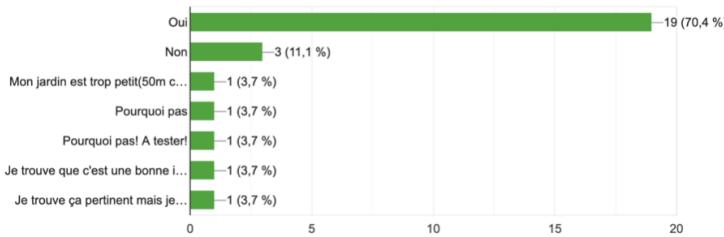
Quels aspects pourraient vous motiver à utiliser cette application régulièrement ?

27 réponses



Pensez-vous que ce projet est pertinent et envisagez-vous d'utiliser ce site ?

27 réponses



Avez-vous des suggestions ou des idées pour améliorer ce projet ?

4 réponses

Voir ma réponse à la question d'avant pour savoir à quoi s'attendre niveau superficie et éventuellement les attentes.

Recherche d'amis connus pour coordination facilitée

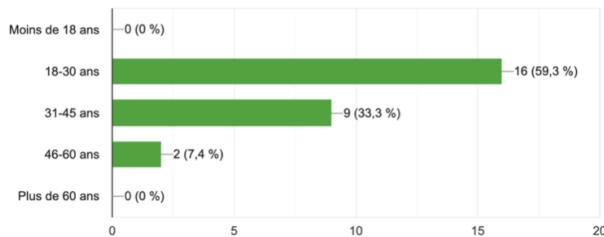
parfait pour des personnes qui ne sont plus capables d'entretenir leur jardin (seniors ou personnes à mobilité réduite)

qu'il y ait un réel échange d'informations, d'apprentissage. former un véritable réseau de passionnées - intérêt ludique, pédagogique et professionnel

Section : Profil général

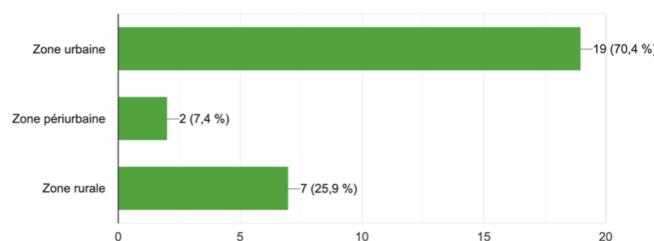
Quel est votre âge ?

27 réponses



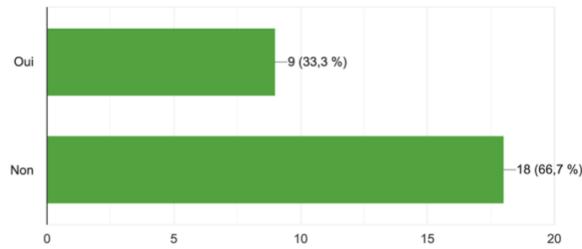
Où habitez-vous ?

27 réponses



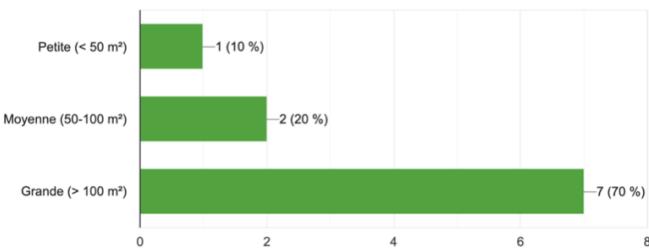
Possédez-vous un jardin ?

27 réponses



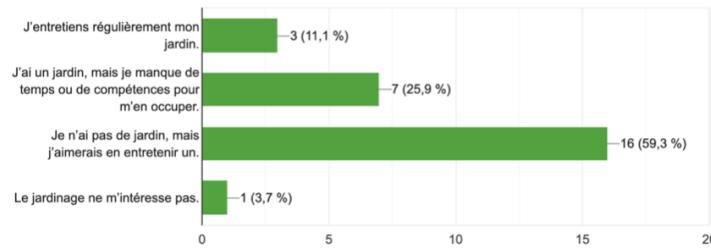
Si oui, quelle est sa taille approximative ?

10 réponses



Quelle est votre relation actuelle avec le jardinage ?

27 réponses



Annexe B – Résultats détaillés du questionnaire (export CSV)

Source : Formulaire JardinSolidaire.csv • 27 réponses • 19 colonnes

Dictionnaire des colonnes

- | | |
|--|--|
| 1. Horodateur | 2. À quelle fréquence entretelez-vous votre jardin ? |
| 3. Quelles tâches nécessitent le plus d'aide dans votre jardin ? | 4. Seriez-vous prêt(e) à accueillir des amis du vert pour vous aider ? |
| 5. Quels critères seraient importants pour choisir un ami du vert ? | 6. Pourquoi aimeriez-vous utiliser une plateforme comme JardinSolidaire ? |
| 7. Quelles compétences avez-vous déjà en jardinage ? | 8. Quelles tâches préférez-vous réaliser en tant qu'ami du vert ? |
| 9. À quelle fréquence seriez-vous disponible pour jardiner ? | 10. Seriez-vous prêt(e) à vous déplacer en dehors de votre zone de résidence pour jardiner ? |
| 11. Quelles sont vos principales préoccupations concernant cette plateforme ? | 12. Quels aspects pourraient vous motiver à utiliser cette application régulièrement ? |
| 13. Pensez-vous que ce projet est pertinent et envisagez-vous d'utiliser ce site ? | 14. Avez-vous des suggestions ou des idées pour améliorer ce projet ? |
| 15. Quel est votre âge ? | 16. Où habitez-vous ? |
| 17. Possédez-vous un jardin ? | 18. Si oui, quelle est sa taille approximative ? |
| 19. Quelle est votre relation actuelle avec le jardinage ? | |

Colonnes affichées — Groupe 1/4

Horodateur	À quelle fréquence entretelez-vous votre jardin ?	Quelles tâches nécessitent le plus d'aide dans votre jardin ?	Seriez-vous prêt(e) à accueillir des amis du vert pour vous aider ?	Quels critères seraient importants pour choisir un ami du vert ?	Pourquoi aimeriez-vous utiliser une plateforme comme JardinSolidaire ?
2025/01/16 4:17:24 PM UTC+1	Occasionnellement, seulement en cas de besoin	Désherbage	Peut-être	Outils disponibles	Accéder à un espace vert pour se détendre
2025/01/16 8:07:54 PM UTC+1					Apprendre le jardinage
2025/01/16 8:16:10 PM UTC+1					Accéder à un espace vert pour se détendre
2025/01/16 8:16:27 PM UTC+1					Apprendre le jardinage
2025/01/16 8:19:53 PM UTC+1	Une fois par mois	Tonte de pelouse	Oui	Avis et recommandations	Accéder à un espace vert pour se détendre
2025/01/16 8:41:26 PM UTC+1					Apprendre le jardinage
2025/01/16 10:30:49 PM UTC+1					Aider des propriétaires de jardins
2025/01/17 3:33:48 PM UTC+1					Apprendre le jardinage
2025/01/17 11:52:49 PM UTC+1	Occasionnellement, seulement en cas de besoin	Taille des haies	Oui	Avis et recommandations	
2025/01/18 9:38:16 AM UTC+1					Accéder à un espace vert pour se détendre
2025/01/18 11:32:21 AM UTC+1	Une fois par semaine	Plantation de fleurs ou légumes	Peut-être	Avis et recommandations	
2025/01/24 10:03:04 AM UTC+1					Accéder à un espace vert pour se détendre
2025/01/24 11:31:37 AM UTC+1	Une fois par mois	Arrosage	Non	Outils disponibles	Apprendre le jardinage
2025/01/24 1:53:22 PM UTC+1					Apprendre le jardinage
2025/01/24 2:22:57 PM UTC+1					Apprendre le jardinage
2025/01/24 4:21:59 PM UTC+1	Une fois par semaine	Désherbage	Non		
2025/01/24 4:41:40 PM UTC+1					Apprendre le jardinage;Accéder à un espace vert pour se détendre;Aider des propriétaires de jardins
2025/01/24 4:45:12 PM UTC+1					Apprendre le jardinage;Accéder à un espace vert pour se détendre
2025/01/24 4:51:47 PM UTC+1	Une fois par mois	Désherbage;Arrosage	Oui	Niveau d'expérience;Avis et recommandations	
2025/01/24 5:23:18 PM UTC+1					Apprendre le jardinage;Accéder à un espace vert pour se détendre;Echanger, partager la passion du jardinage avec les propriétaires
2025/01/24 9:14:10 PM UTC+1	Occasionnellement, seulement en cas de besoin	Tonte de pelouse	Peut-être	Outils disponibles	
2025/01/25 7:50:33 AM UTC+1	Occasionnellement, seulement en cas de besoin	Taille des haies	Peut-être	La bienveillance	
2025/01/25 8:12:28 AM UTC+1	Occasionnellement, seulement en cas de besoin	Arrosage	Peut-être	Avis et recommandations	Apprendre le jardinage;Accéder à un espace vert pour se détendre

Horodateur	À quelle fréquence entretelez-vous votre jardin ?	Quelles tâches nécessitent le plus d'aide dans votre jardin ?	Seriez-vous prêt(e) à accueillir des amis du vert pour vous aider ?	Quels critères seraient importants pour choisir un ami du vert ?	Pourquoi aimeriez-vous utiliser une plateforme comme JardinSolidaire ?
2025/01/25 8:50:20 PM UTC+1					Aider des propriétaires de jardins
2025/01/26 11:26:39 AM UTC+1	Occasionnellement, seulement en cas de besoin	Désherbage;Plantation de fleurs ou légumes	Oui	Niveau d'expérience;Avis et recommandations	
2025/01/26 2:04:53 PM UTC+1	Occasionnellement, seulement en cas de besoin	Tonte de pelouse;Désherbage;Taille des haies;Arrosage;ramassage de feuilles mortes	Peut-être	Niveau d'expérience;Avis et recommandations;Outils disponibles;relation de confiance	Apprendre le jardinage
2025/02/04 8:20:43 PM UTC+1					Accéder à un espace vert pour se détendre

Colonnes affichées — Groupe 2/4

Horodateur	Quelles compétences avez-vous déjà en jardinage ?	Quelles tâches préférez-vous réaliser en tant qu'amis du vert ?	À quelle fréquence seriez-vous disponible pour jardiner ?	Seriez-vous prêt(e) à vous déplacer en dehors de votre zone de résidence pour jardiner ?	Quelles sont vos principales préoccupations concernant cette plateforme ?
2025/01/16 4:17:24 PM UTC+1	Aucune, je suis débutant(e).	Arrosage	Occasionnellement, seulement en cas de besoin	Cela dépend de la distance.	Difficulté d'utilisation de la plateforme
2025/01/16 8:07:54 PM UTC+1	Aucune, je suis débutant(e). Quelques bases (tonte, désherbage, etc.).	tous	Une fois par mois; Occasionnellement, seulement en cas de besoin	Oui; Cela dépend de la distance.	Qualité des interventions (compétences des amis du vert)
2025/01/16 8:16:10 PM UTC+1	Aucune, je suis débutant(e).	Arrosage	Une fois par semaine	Non	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/16 8:16:27 PM UTC+1	Aucune, je suis débutant(e).	Arrosage	Occasionnellement, seulement en cas de besoin	Non	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/16 8:19:53 PM UTC+1	Aucune, je suis débutant(e).	Plantation	Occasionnellement, seulement en cas de besoin	Non	Difficulté d'utilisation de la plateforme
2025/01/16 8:41:26 PM UTC+1	Aucune, je suis débutant(e).	Tonte	Occasionnellement, seulement en cas de besoin	Cela dépend de la distance.	Qualité des interventions (compétences des amis du vert)
2025/01/16 10:30:49 PM UTC+1	Quelques bases (tonte, désherbage, etc.).	Tonte	Une fois par mois	Cela dépend de la distance.	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/17 3:33:48 PM UTC+1	Aucune, je suis débutant(e).	Plantation	Une fois par mois	Oui	Qualité des interventions (compétences des amis du vert)
2025/01/17 11:52:49 PM UTC+1					Sécurité des échanges (fiabilité des utilisateurs)
2025/01/18 9:38:16 AM UTC+1	Quelques bases (tonte, désherbage, etc.).	Tonte	Une fois par mois	Oui	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/18 11:32:21 AM UTC+1					Sécurité des échanges (fiabilité des utilisateurs)
2025/01/24 10:03:04 AM UTC+1	Quelques bases (tonte, désherbage, etc.).	Plantation	Une fois par semaine	Cela dépend de la distance.	Qualité des interventions (compétences des amis du vert)
2025/01/24 11:31:37 AM UTC+1	Quelques bases (tonte, désherbage, etc.).	Arrosage	Une fois par mois	Non	Qualité des interventions (compétences des amis du vert)
2025/01/24 1:53:22 PM UTC+1	Aucune, je suis débutant(e).	Plantation	Une fois par semaine	Cela dépend de la distance.	Confidentialité des données personnelles
2025/01/24 2:22:57 PM UTC+1	Aucune, je suis débutant(e).	Plantation	Une fois par semaine	Cela dépend de la distance.	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/24 4:21:59 PM UTC+1					Sécurité des échanges (fiabilité des utilisateurs)
2025/01/24 4:41:40 PM UTC+1	Quelques bases (tonte, désherbage, etc.).	Plantation; Arrosage	Une fois par semaine	Cela dépend de la distance.	Confidentialité des données personnelles
2025/01/24 4:45:12 PM UTC+1	Aucune, je suis débutant(e).	Plantation; Arrosage	Occasionnellement, seulement en cas de besoin	Cela dépend de la distance.	Sécurité des échanges (fiabilité des utilisateurs)
2025/01/24 4:51:47 PM UTC+1					Sécurité des échanges (fiabilité des utilisateurs); Confidentialité des données personnelles; Qualité des interventions (compétences des amis du vert); Difficulté d'utilisation de la plateforme

Horodateur	Quelles compétences avez-vous déjà en jardinage ?	Quelles tâches préférez-vous réaliser en tant qu'amis du vert ?	À quelle fréquence seriez-vous disponible pour jardiner ?	Seriez-vous prêt(e) à vous déplacer en dehors de votre zone de résidence pour jardiner ?	Quelles sont vos principales préoccupations concernant cette plateforme ?
2025/01/24 5:23:18 PM UTC+1	Quelques bases (tonte, désherbage, etc.).	Plantation; Arrosage; Désherbage; Taille des haies; Diagnostic? Construction (mare, mur, pas japonais...)	Une fois par mois	Cela dépend de la distance.	Sécurité des échanges (fiabilité des utilisateurs); peut être créer du lien avec d'autre amis du vert comme des "chantiers participatifs"
2025/01/24 9:14:10 PM UTC+1					Sécurité des échanges (fiabilité des utilisateurs); Confidentialité des données personnelles
2025/01/25 7:50:33 AM UTC+1					Sécurité des échanges (fiabilité des utilisateurs); Qualité des interventions (compétences des amis du vert)
2025/01/25 8:12:28 AM UTC+1	Aucune, je suis débutant(e).	Plantation; Taille des haies	Une fois par mois	Cela dépend de la distance.	Qualité des interventions (compétences des amis du vert)
2025/01/25 8:50:20 PM UTC+1	Avancé(e), je peux gérer des tâches complexes.	Plantation; Arrosage; Désherbage; Tonte	Une fois par semaine	Cela dépend de la distance.	Sécurité des échanges (fiabilité des utilisateurs); Confidentialité des données personnelles; Qualité des interventions (compétences des amis du vert)
2025/01/26 11:26:39 AM UTC+1					Sécurité des échanges (fiabilité des utilisateurs); Qualité des interventions (compétences des amis du vert)
2025/01/26 2:04:53 PM UTC+1	Quelques bases (tonte, désherbage, etc.).	Plantation	Occasionnellement, seulement en cas de besoin	Non	Sécurité des échanges (fiabilité des utilisateurs); Confidentialité des données personnelles; Difficulté d'utilisation de la plateforme
2025/02/04 8:20:43 PM UTC+1	Avancé(e), je peux gérer des tâches complexes.	Plantation; Arrosage	Une fois par mois	Oui	Sécurité des échanges (fiabilité des utilisateurs); Confidentialité des données personnelles

Colonnes affichées — Groupe 3/4

Horodateur	Quels aspects pourraient vous motiver à utiliser cette application régulièrement ?	Pensez-vous que ce projet est pertinent et envisagez-vous d'utiliser ce site ?	Avez-vous des suggestions ou des idées pour améliorer ce projet ?	Quel est votre âge ?	Où habitez-vous ?
2025/01/16 4:17:24 PM UTC+1	Communauté active et fiable	Oui		18-30 ans	Zone rurale
2025/01/16 8:07:54 PM UTC+1	Communauté active et fiable	Oui		18-30 ans	Zone urbaine
2025/01/16 8:16:10 PM UTC+1	Communauté active et fiable	Oui		31-45 ans	Zone urbaine
2025/01/16 8:16:27 PM UTC+1	Facilité d'utilisation	Pourquoi pas		18-30 ans	Zone urbaine
2025/01/16 8:19:53 PM UTC+1	Communauté active et fiable	Oui		46-60 ans	Zone urbaine
2025/01/16 8:41:26 PM UTC+1	Communauté active et fiable	Oui		31-45 ans	Zone urbaine
2025/01/16 10:30:49 PM UTC+1	Communauté active et fiable	Oui		18-30 ans	Zone rurale
2025/01/17 3:33:48 PM UTC+1	Large choix de jardins ou d'amis du vert	Oui		18-30 ans	Zone urbaine
2025/01/17 11:52:49 PM UTC+1	Facilité d'utilisation	Oui		31-45 ans	Zone rurale
2025/01/18 9:38:16 AM UTC+1	Communauté active et fiable	Oui		18-30 ans	Zone urbaine
2025/01/18 11:32:21 AM UTC+1	Communauté active et fiable	Non		31-45 ans	Zone rurale
2025/01/24 10:03:04 AM UTC+1	Communauté active et fiable	Oui		18-30 ans	Zone périurbaine
2025/01/24 11:31:37 AM UTC+1	Facilité d'utilisation	Non		18-30 ans	Zone urbaine
2025/01/24 1:53:22 PM UTC+1	Large choix de jardins ou d'amis du vert	Je trouve ça pertinent mais je me vois pas l'utiliser		18-30 ans	Zone urbaine
2025/01/24 2:22:57 PM UTC+1	Système d'échange équitable	Oui		18-30 ans	Zone urbaine
2025/01/24 4:21:59 PM UTC+1	Communauté active et fiable	Non	parfait pour des personnes qui ne sont plus capables d'entretenir leur jardin (seniors ou personnes à mobilité réduite)	31-45 ans	Zone urbaine
2025/01/24 4:41:40 PM UTC+1	Communauté active et fiable; Large choix de jardins ou d'amis du vert	Oui		18-30 ans	Zone périurbaine
2025/01/24 4:45:12 PM UTC+1	Facilité d'utilisation; Communauté active et fiable; Large choix de jardins ou d'amis du vert	Oui		18-30 ans	Zone urbaine
2025/01/24 4:51:47 PM UTC+1	Facilité d'utilisation; Communauté active et fiable	Oui		18-30 ans	Zone rurale
2025/01/24 5:23:18 PM UTC+1	Communauté active et fiable; Système d'échange équitable	Je trouve que c'est une bonne idée mais il manque un côté "convivial" au projet	qu'il y ait un réel échange d'informations, d'apprentissage, former un véritable réseau de passionnées - intérêt ludique, pédagogique et professionnel	18-30 ans	Zone urbaine
2025/01/24 9:14:10 PM UTC+1	Facilité d'utilisation; Communauté active et fiable	Pourquoi pas! A tester!		31-45 ans	Zone urbaine
2025/01/25 7:50:33 AM UTC+1	Facilité d'utilisation; Communauté active et fiable; Large choix de jardins ou d'amis du vert; Système d'échange équitable	Mon jardin est trop petit(50m carré) mais le projet est pertinent à partir d'une certaine superficie ou dépendance des habitant(e)s	Voir ma réponse à la question d'avant pour savoir à quoi s'attendre niveau superficie et éventuellement les attentes.	31-45 ans	Zone urbaine
2025/01/25 8:12:28 AM UTC+1	Communauté active et fiable; Large choix de jardins ou d'amis du vert	Oui		18-30 ans	Zone urbaine

Horodateur	Quels aspects pourraient vous motiver à utiliser cette application régulièrement ?	Pensez-vous que ce projet est pertinent et envisagez-vous d'utiliser ce site ?	Avez-vous des suggestions ou des idées pour améliorer ce projet ?	Quel est votre âge ?	Où habitez-vous ?
2025/01/25 8:50:20 PM UTC+1	Communauté active et fiable; Large choix de jardins ou d'amis du vert; Système d'échange équitable	Oui		18-30 ans	Zone urbaine
2025/01/26 11:26:39 AM UTC+1	Communauté active et fiable	Oui		31-45 ans	Zone rurale
2025/01/26 2:04:53 PM UTC+1	Communauté active et fiable; Large choix de jardins ou d'amis du vert; Système d'échange équitable	Oui	Recherche d'amis connus pour coordination facilitée	31-45 ans	Zone urbaine; Zone rurale
2025/02/04 8:20:43 PM UTC+1	Communauté active et fiable; Large choix de jardins ou d'amis du vert; Système d'échange équitable	Oui		46-60 ans	Zone urbaine

Colonnes affichées — Groupe 4/4

Horodateur	Possédez-vous un jardin ?	Si oui, quelle est sa taille approximative ?	Quelle est votre relation actuelle avec le jardinage ?
2025/01/16 4:17:24 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/16 8:07:54 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/16 8:16:10 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/16 8:16:27 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/16 8:19:53 PM UTC+1	Non	Grande (> 100 m ²)	J'entretiens régulièrement mon jardin.
2025/01/16 8:41:26 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/16 10:30:49 PM UTC+1	Oui	Grande (> 100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/17 3:33:48 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/17 11:52:49 PM UTC+1	Oui	Grande (> 100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/18 9:38:16 AM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/18 11:32:21 AM UTC+1	Oui	Grande (> 100 m ²)	J'entretiens régulièrement mon jardin.
2025/01/24 10:03:04 AM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 11:31:37 AM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 1:53:22 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 2:22:57 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 4:21:59 PM UTC+1	Oui	Moyenne (50-100 m ²)	J'entretiens régulièrement mon jardin.
2025/01/24 4:41:40 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 4:45:12 PM UTC+1	Non		Le jardinage ne m'intéresse pas.
2025/01/24 4:51:47 PM UTC+1	Oui	Grande (> 100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/24 5:23:18 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/24 9:14:10 PM UTC+1	Oui	Petite (< 50 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/25 7:50:33 AM UTC+1	Oui	Moyenne (50-100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/25 8:12:28 AM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/25 8:50:20 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.
2025/01/26 11:26:39 AM UTC+1	Oui	Grande (> 100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/01/26 2:04:53 PM UTC+1	Oui	Grande (> 100 m ²)	J'ai un jardin, mais je manque de temps ou de compétences pour m'en occuper.
2025/02/04 8:20:43 PM UTC+1	Non		Je n'ai pas de jardin, mais j'aimerais en entretenir un.

Annexe C — Cahier des charges

Cahier des charges (document complet de la première version faite à trois au commencement du projet).

Cahier des charges

JardinSolidaire

Contact des personnes en charge du projet :

Nom : Woods

Poste :

Prénom : Thalia

Mail : thaliadwoods@gmail.com

Nom : Vernon

Poste :

Prénom : Hanaë

Mail : vernonhanae@gmail.com

Nom : Bouillart

Poste :

Prénom : Médina

Mail : medina.bllrt@gmail.com

Sommaire

1. Contexte et objectifs du projet

- 1.1 Présentation du site
- 1.2 Identité de l'entreprise
- 1.3 Valeurs de l'entreprise
- 1.4 Description des services / produits
- 1.5 Nos principaux concurrents hors internet
- 1.6 Nos principaux concurrents sur internet
- 1.7 Quels sont vos partenaires
- 1.8 Nombre de pages envisagées

2. Caractéristiques graphiques

- 2.1 Couleurs
- 2.2 Polices de caractère
- 2.3 Goûts et préférences
- 2.4 Maquette de site – Web Design

3. Objectifs

- 3.1 Objectifs qualitatifs
- 3.2 Objectifs quantitatifs
- 3.3 La cible

4. Description des besoins fonctionnels

- 4.1 Fonctionnalités principales
- 4.2 Éléments du menu
- 4.3 Scénarios utilisateur·ice

5. Contraintes techniques et organisationnelles

- 5.1 Contraintes techniques
- 5.2 Contraintes organisationnelles

6. Planning et étapes clés

7. Ressources nécessaires

- 7.1 Ressources humaines
- 7.2 Ressources techniques
- 7.3 Ressources matérielles et financières

8. Sécurité et accessibilité

- 8.1 Sécurité
- 8.2 Accessibilité

1 . Contexte et objectifs du projet

Objectif : Créer un site web accessible sur mobile, tablette et ordinateur, qui permet :

- Aux propriétaires de jardins de publier des annonces pour trouver des "amis du vert" prêts à entretenir leurs espaces verts.
- Aux "amis du vert" de consulter les jardins disponibles, réserver des créneaux, et contribuer à leur entretien.
- De faciliter les échanges entre utilisateurs grâce à une messagerie intégrée et des outils interactifs comme un calendrier et des filtres de recherche.

Le site devra être intuitif, sécurisé, et offrir une expérience utilisateur fluide, adaptée à tous les profils.

1.1 Présentation du site

Nom de l'entreprise : JardinSolidaire

Date de création du projet :

Nom de domaine : jardinsolidaire.fr

Secteur d'activité : SAS Application Web dans la mise en relation de service de jardinage
Application web de mise en relation de jardinage

Il s'agit d'une création de projet

1.2 Identité de l'entreprise

Nom de l'entreprise : JardinSolidaire

Date de création : 06/01/2025

Nombre de salariés : 3

Activité principale : JardinSolidaire est une plateforme collaborative conçue pour connecter les propriétaires de jardins privés, souvent sous-utilisés, avec des amateurs de nature, appelés 'amis du vert'. L'objectif est de favoriser l'entretien des espaces verts tout en offrant aux participants une opportunité unique de partager, apprendre, et profiter de ces lieux. Grâce à un système d'échange basé sur le temps passé dans les jardins plutôt que sur une transaction financière, JardinSolidaire valorise le partage, la solidarité et la durabilité, tout en créant du lien social autour de la nature.

1.3 Valeurs de l'entreprise

Slogan potentiel

"Cultivons ensemble : partage, solidarité, nature."

Message principal

JardinSolidaire incarne une vision d'entraide et de partage, où les jardins privés deviennent des lieux collaboratifs, favorisant à la fois la connexion humaine et la préservation de la nature.

Valeurs clés

1. **Partage** : Valoriser les ressources disponibles en créant des échanges non financiers entre propriétaires de jardins et amis du vert.
2. **Solidarité** : Renforcer les liens sociaux, en particulier pour les personnes isolées, à travers une dynamique d'entraide locale.
3. **Durabilité** : Encourager des pratiques respectueuses de l'environnement pour préserver les espaces verts et la biodiversité.
4. **Accessibilité** : Proposer une plateforme gratuite et ouverte à tous, indépendamment des moyens financiers ou des compétences en jardinage.
5. **Collaboration** : Fédérer une communauté autour d'un projet commun, où chaque participant contribue activement à l'entretien et à la valorisation des jardins.

1.4 Description des services / produits

Le projet **JardinSolidaire** consiste à développer une plateforme en ligne dédiée à la mise en relation entre propriétaires de jardins privés et passionnés de jardinage ("amis du vert"). Cette plateforme offre les services suivants :

1. **Mise en relation**
 - Les propriétaires de jardins peuvent publier des annonces décrivant leurs espaces verts et leurs besoins spécifiques (entretien, plantation, désherbage, etc.).
 - Les amis du vert peuvent rechercher des jardins à proximité et postuler pour contribuer à leur entretien.
2. **Réservation simplifiée**
 - Un système de calendrier interactif permet aux utilisateurs de définir leurs disponibilités et de réserver des créneaux pour des sessions de jardinage.
3. **Communication directe**
 - Une messagerie intégrée facilite les échanges entre les propriétaires et les amis du vert pour organiser les détails des sessions.
4. **Évaluation et retours d'expérience**
 - Les utilisateurs peuvent laisser des avis sur leurs interactions pour renforcer la confiance au sein de la communauté.
5. **Accessibilité multiplateforme**
 - Le site sera optimisé pour une utilisation fluide sur mobile, tablette et ordinateur, garantissant une accessibilité pour tous les profils d'utilisateurs.

Cette plateforme repose sur un modèle entièrement gratuit, mettant en avant l'échange de temps et la solidarité, sans transaction financière.

1.5 Nos principaux concurrents hors internet

Bien que **JardinSolidaire** soit principalement une plateforme numérique, il existe des alternatives hors internet qui répondent partiellement aux besoins des propriétaires de jardins et des passionnés de nature. Ces concurrents traditionnels incluent :

1. Services de jardinage professionnels

- **Description** : Des entreprises ou des jardiniers indépendants proposent des services d'entretien des jardins (tonte, désherbage, plantation, etc.).
- **Forces**:
 - Prestation professionnelle et garantie de qualité.
 - Disponibilité dans de nombreuses régions.
- **Faiblesses**:
 - Coût élevé, souvent inaccessible pour les foyers modestes.
 - Absence de dimension collaborative ou sociale.

2. Échanges informels dans les quartiers

- **Description** : Les propriétaires de jardins peuvent solliciter des voisins, amis ou membres de leur famille pour les aider à entretenir leurs espaces verts.
- **Forces**:
 - Confiance préétablie entre les participants.
 - Gratuité ou échanges non formalisés (service contre service).
- **Faiblesses**:
 - Portée limitée : dépend de la proximité géographique et des relations personnelles.
 - Absence de structure ou de cadre organisationnel.

3. Associations locales de jardinage

- **Description** : Certaines associations organisent des ateliers, des échanges de services ou des projets communautaires autour du jardinage.
- **Forces**:
 - Dynamique locale et possibilité d'apprentissage.
 - Sensibilisation à des pratiques écologiques et collaboratives.
- **Faiblesses**:
 - Peu d'associations disponibles dans certaines zones.
 - Public limité à un réseau restreint d'adhérents.

4. Nos principaux concurrents sur internet

Sur Internet, plusieurs plateformes existent dans le domaine du jardinage, de la location d'espaces verts ou des échanges collaboratifs. Bien que ces concurrents ne proposent pas exactement le même service que **JardinSolidaire**, ils partagent certaines similitudes en termes de mise en relation ou d'objectifs environnementaux.

1. JardinPrivé.fr

- **Description :** Plateforme permettant aux propriétaires de louer leur jardin pour des événements privés ou des loisirs.
- **Forces:**
 - Large éventail d'offres et présentation claire des annonces.
 - Mise en avant des jardins comme espaces de détente.
- **Faiblesses:**
 - Service payant, ce qui peut limiter l'accès pour certains utilisateurs.
 - Absence de dimension collaborative ou de mise en valeur du jardinage.

2. Privateaser

- **Description :** Site spécialisé dans la réservation d'espaces privés, incluant parfois des jardins, pour des événements (fêtes, réunions, etc.).
- **Forces:**
 - Fonctionnalités avancées de recherche et de filtrage pour trouver un lieu adapté.
 - Audience large grâce à une spécialisation événementielle.
- **Faiblesses:**
 - Focalisation sur les événements sans lien direct avec le jardinage.
 - Service orienté vers des clients ayant un budget conséquent.

5. Quels sont vos partenaires

Bien que le projet JardinSolidaire ne soit pas encore lancé, plusieurs types de partenaires potentiels pourraient contribuer à son développement et à sa pérennité. Ces collaborations permettraient de renforcer l'impact du projet tout en élargissant son réseau et ses capacités.

1. Partenaires institutionnels

- Collectivités locales et régionales:
 - Soutien logistique et promotion locale de la plateforme.
 - Organisation d'événements pour sensibiliser les habitants à l'entraide et à la préservation des espaces verts.
- Associations environnementales:
 - Sensibilisation à des pratiques de jardinage respectueuses de l'environnement.
 - Partage de ressources pédagogiques pour encourager la biodiversité.

2. Établissements éducatifs et sociaux

- Rôle potentiel:
 - Encourager les jeunes ou les groupes marginalisés (personnes âgées, en insertion sociale, etc.) à participer à des activités collaboratives.
 - Utiliser la plateforme comme outil pédagogique pour sensibiliser aux enjeux environnementaux.
- Exemples : Lycées agricoles, centres sociaux, maisons de quartier.

Ces partenaires potentiels permettront d'assurer une visibilité et un impact durables pour **JardinSolidaire**, tout en renforçant son positionnement dans l'économie solidaire et écologique.

2 . Caractéristiques Graphiques

2.1. Couleurs

Voici les couleurs proposées, adaptées à différentes utilisations :



- **Couleur principale : Pistachio (A7D477)**
Utilisation : Fond des sections, zones principales.
- **Couleur secondaire : Mexican Pink (E3107D)**
Utilisation : Boutons, appels à l'action.
- **Couleur d'accentuation : Dark Green (021904)**
Utilisation : Titres, liens, éléments interactifs.
- **Couleur neutre : Lavender Blush (FFF0F0)**
Utilisation : Fond global.
- **Couleur complémentaire : Mindaro (E4F1AC)**
Utilisation : Hover ou mise en avant discrète.

2.2 Les polices de caractère

Pour une bonne lisibilité et un style cohérent, voici une combinaison de polices :

1. **Titre : Montserrat**
Style moderne et épuré, idéal pour les titres. (Google Fonts)
2. **Sous-titres : Lato**
Bonne lisibilité pour des textes intermédiaires.
3. **Corps de texte : Open Sans**
Une police simple et universelle pour garantir la lisibilité sur tous les navigateurs.
4. **Boutons : Poppins**
Style dynamique et impactant pour les appels à l'action.

2.3. Goûts et préférences

Sites Internet qui correspondent à nos attentes

1. **JardinPrivé.fr**

Adresse : <https://www.jardinprive.fr>

Ce qui nous plaît :

- Présentation claire et simple des jardins disponibles.
- Mise en avant des images pour attirer l'attention.
- Système de recherche intuitif avec filtres.

2. Airbnb.fr

Adresse : <https://www.airbnb.fr>

Ce qui nous plaît :

- Design moderne et épuré.
- Interface utilisateur intuitive, avec une navigation fluide.
- Système de réservation clair et efficace, renforçant la confiance entre utilisateurs.

3. PlantezChezNous.com

Adresse : <https://www.plantezcheznous.com>

Ce qui nous plaît :

- Concept axé sur le partage des espaces verts.
- Approche collaborative mettant en avant l'entraide.
- Mise en valeur des annonces avec des descriptions détaillées.

Sites Internet qui ne nous plaisent pas

1. Sites avec un design obsolète

- **Ce que nous n'aimons pas:**
 - Interfaces visuellement datées et peu engageantes.
 - Manque de modernité dans la mise en page, souvent peu adaptée aux écrans actuels.

2. Sites non accessibles

- **Ce que nous n'aimons pas:**
 - Absence de responsive design, rendant l'expérience difficile sur mobile ou tablette.
 - Navigation peu intuitive, compliquant l'accès aux informations essentielles.

3. Sites avec une mauvaise expérience utilisateur

- **Ce que nous n'aimons pas:**
 - Trop d'éléments inutiles ou mal organisés, créant une confusion pour l'utilisateur.
 - Temps de chargement trop longs, provoquant de la frustration.
 - Absence de filtres ou d'outils pour affiner les recherches, ce qui allonge inutilement la navigation.

2.4. Maquette de site - Web Design

Outils utilisés

1. **Miro :**
 - Utilisé pour créer les **workflows utilisateurs**, afin d'illustrer les parcours des deux profils principaux :
 - **Propriétaires de jardins** : depuis la création de compte jusqu'à la gestion des annonces.
 - **Amis du vert** : depuis la recherche d'un jardin jusqu'à la réservation et l'évaluation.
 - Les workflows montrent les différentes étapes d'interaction, facilitant la compréhension des besoins fonctionnels.
2. **Figma :**
 - Utilisé pour concevoir les **wireframes** (maquettes basse fidélité) et les maquettes responsives.
 - Permet de visualiser l'apparence et la structure des principales pages du site sur différents supports (mobile, tablette, desktop).
 - Inclut des composants réutilisables pour garantir la cohérence visuelle et fonctionnelle.

Pages principales modélisées

1. **Page d'accueil :**
 - Présentation des fonctionnalités principales (recherche, accès au profil, messagerie).
 - Mise en avant des annonces récentes et jardins populaires.
2. **Page de recherche :**
 - Barre de recherche avec filtres (localisation, besoins, type de jardin).
 - Résultats présentés sous forme de cartes ou de listes avec des photos et descriptions.
3. **Page de profil utilisateur :**
 - Informations personnelles (photo, description).
 - Section pour gérer les annonces (propriétaires) ou consulter les réservations (amis du vert).
4. **Calendrier interactif :**
 - Gestion des créneaux disponibles pour les propriétaires.
 - Consultation et réservation de créneaux pour les amis du vert.
5. **Messagerie :**
 - Interface pour échanger directement entre utilisateurs.
 - Notifications et historique des conversations.

Livrables associés

1. **Workflows** (réalisés sur Miro) :
 - Diagrammes détaillant les étapes et interactions des utilisateurs.
 - Identification des points clés du parcours pour optimiser l'expérience utilisateur.
2. **Wireframes et maquettes responsives** (réalisés sur Figma) :
 - Mobile : Design simplifié avec navigation compacte.
 - Tablette : Mise en page optimisée pour une utilisation intermédiaire.
 - Desktop : Interface complète et structurée pour une navigation claire.

2.5 Nombre de pages envisagées

Pages publiques (accessibles sans connexion)

- Page d'accueil** : Présentation du projet, des fonctionnalités principales, et des appels à l'action (inscription ou connexion).
- Page de connexion/inscription** : Formulaire pour créer un compte ou accéder à un compte existant.
- Page "À propos"** : Description du projet, de ses valeurs, et de son fonctionnement.

Pages privées (accessibles après connexion)

- Tableau de bord utilisateur** : Vue d'ensemble des interactions, des annonces, et des missions.
- Page de recherche de jardins** : Liste des jardins disponibles avec filtres et barre de recherche.
- Page de recherche de jardiniers** : Liste des amis du vert disponibles, avec profils et compétences.
- Page de publication de jardins** : Formulaire pour ajouter un jardin avec description, photos, et calendrier.
- Page de profil utilisateur** : Informations personnelles, historique des interactions, et heures cumulées.
- Messagerie** : Interface de communication entre utilisateurs (propriétaires et amis du vert).
- Calendrier interactif** : Consultation et gestion des créneaux disponibles ou réservés.
- Page de réservation** : Détail des réservations à venir ou passées, avec possibilité de laisser un avis.

Pages supplémentaires

- Page FAQ et assistance** : Réponses aux questions fréquentes et formulaire de contact.
- Page "Conditions générales d'utilisation"** : Règles et informations légales liées à l'utilisation de la plateforme.

Total : 13 pages principales

3 . Objectifs

3.1 Objectifs qualitatifs

Les objectifs qualitatifs de JardinSolidaire reflètent la volonté de créer un impact durable et positif sur les utilisateurs et la société. Ces objectifs ne sont pas mesurables directement par des chiffres, mais ils orientent la conception et la gestion du projet.

1. Améliorer l'image des espaces verts privés

- Valoriser les jardins sous-utilisés en les transformant en lieux de collaboration et de partage.
- Renforcer l'idée que les jardins peuvent être des espaces communs bénéfiques à tous.

2. Promouvoir la solidarité et le lien social

- Encourager les échanges entre des individus de différents horizons, en favorisant une dynamique d'entraide.

- Réduire l'isolement social, notamment chez les personnes âgées ou vivant seules.

3. Offrir une expérience utilisateur fluide et accessible

- Proposer une plateforme intuitive qui simplifie les échanges et les interactions.
- Garantir une accessibilité adaptée à tous les profils, même ceux peu familiers avec les outils numériques.

4. Sensibiliser à des pratiques durables

- Encourager les utilisateurs à adopter des méthodes de jardinage respectueuses de l'environnement.
- Renforcer la prise de conscience écologique et l'importance de la préservation des espaces verts.

5. Assurer la satisfaction des utilisateurs

- Construire une plateforme qui répond réellement aux attentes des propriétaires de jardins et des amis du vert.
- Créer un environnement où les utilisateurs se sentent valorisés et encouragés à participer activement.

3.2 Objectifs quantitatifs

Les objectifs quantitatifs de JardinSolidaire permettent de mesurer la réussite du projet à travers des indicateurs précis et concrets. Ces objectifs couvrent différents aspects, notamment le trafic, l'engagement des utilisateurs, et les ressources financières.

1. Trafic et utilisation du site

- Atteindre 100 visites uniques par mois dans les 6 premiers mois après le lancement.
- Avoir une durée moyenne de session d'au moins 3 minutes par utilisateur.
- Obtenir un taux de rebond inférieur à 40 %, indiquant une expérience utilisateur satisfaisante.

2. Nombre d'inscriptions

- Enregistrer 90 comptes utilisateurs actifs (propriétaires de jardins et amis du vert) dans les 6 premiers mois.
- Avoir au moins 30 % de propriétaires de jardins parmi les utilisateurs inscrits.

3. Publications et réservations

- Recevoir 50 annonces de jardins publiées dans les 3 premiers mois.
- Atteindre 10 réservations de créneaux via le calendrier interactif dans les 3 premiers mois.

4. Satisfaction des utilisateurs

- Obtenir un score moyen de 4/5 sur les avis laissés par les utilisateurs.
- Assurer que 80 % des utilisateurs recommandent la plateforme à leur entourage.

5. Objectifs financiers

- Maintenir un budget initial de 20 € pour le développement et le lancement du site (hébergement, design, développement, marketing).
- Limiter les coûts mensuels à 20 € pour l'entretien technique et les campagnes de promotion.

3.3 La cible

1. Propriétaires de jardins

Les propriétaires de jardins constituent la première cible principale de **JardinSolidaire**. Ils représentent les utilisateurs qui mettent leurs espaces verts à disposition sur la plateforme.

- **Profil type :**

- Âge : 40 ans et plus, avec une majorité de 50-65 ans (d'après les statistiques, 70,2 % des plus de 65 ans possèdent un jardin).
- Localisation : Zones périurbaines ou rurales, avec de grands jardins souvent sous-utilisés.
- Besoins : Trouver une aide pour l'entretien de leur jardin (désherbage, tonte, plantation).
- Motivations :
 - Valoriser leur jardin sans engager de frais financiers.
 - Rencontrer des personnes partageant un intérêt pour la nature.
 - Contribuer à une dynamique solidaire et écologique.
- Freins potentiels : Méfiance envers les inconnus ou peur de ne pas trouver quelqu'un de compétent.

2. Amis du vert

Les amis du vert sont les jardiniers amateurs ou passionnés de nature cherchant un espace pour exercer leurs activités en plein air.

- **Profil type :**

- Âge : 25 à 50 ans, avec une proportion significative de citadins sans accès à un jardin.
- Localisation : Principalement des zones urbaines ou périurbaines.
- Besoins : Trouver un espace vert pour se détendre, jardiner, ou apprendre de nouvelles compétences.
- Motivations :
 - Se reconnecter à la nature et s'impliquer dans des activités manuelles.
 - Profiter d'un jardin en échange de leur temps et de leurs compétences.
 - Découvrir des pratiques écologiques et enrichir leurs connaissances.
- Freins potentiels : Accessibilité des jardins (distance géographique) ou crainte de ne pas être à la hauteur des attentes des propriétaires.

3. Populations vulnérables ou spécifiques

- **Personnes âgées isolées** : Besoin d'aide pour entretenir leur jardin tout en recherchant une interaction sociale.

- **Familles monoparentales** : Souhaitant offrir à leurs enfants un accès à la nature tout en gérant un budget limité.
- **Jeunes adultes ou étudiants** : Intéressés par des échanges enrichissants et des expériences pratiques en jardinage.

4 . Description des besoins fonctionnels

4.1 Définir les fonctionnalités principales

1. Création de compte

- Permet aux utilisateurs de s'inscrire en tant que **propriétaire de jardin ou ami du vert**.
- Sécurisation via double authentification par e-mail.
- Options de connexion rapide via Google, Apple ou Facebook.

2. Publication et gestion des jardins (*pour les propriétaires*)

- Ajout d'un jardin avec une description complète (superficie, type de jardin, besoins spécifiques).
- Téléchargement de photos pour illustrer le jardin.
- Gestion des disponibilités via un calendrier interactif.

3. Recherche de jardins et de missions (*pour les amis du vert*)

- Barre de recherche avec filtres avancés :
 - Localisation.
 - Type de jardin (potager, fleuri, mixte).
 - Besoins spécifiques (désherbage, plantation, etc.).
 - Superficie du jardin.
 - Notes et avis des autres utilisateurs.
- Géolocalisation pour trouver des jardins à proximité.

4. Réservation et planification

- Système de réservation pour les amis du vert, permettant de choisir un créneau sur le calendrier du propriétaire.
- Notifications automatiques pour confirmer, annuler ou modifier une réservation.

5. Messagerie intégrée

- Communication directe entre propriétaires et amis du vert.
- Notifications des nouveaux messages pour une meilleure réactivité.

6. Système de notation et d'avis

- Les utilisateurs peuvent évaluer leur expérience après chaque interaction.
- Les avis sont visibles sur les profils pour renforcer la confiance.

7. Gestion des heures cumulées (*pour les amis du vert*)

- Suivi des heures travaillées dans les jardins.
- Possibilité d'utiliser ces heures pour réserver du temps dans un autre jardin.

8. Page de profil utilisateur

- Informations personnelles (photo, description, type d'utilisateur).
- Historique des interactions (missions passées, jardins visités).
- Statistiques personnelles (heures cumulées, évaluations, etc.).

9. Accessibilité multiplateforme

- Site responsive, optimisé pour les écrans mobiles, tablettes et ordinateurs.
- Interface utilisateur intuitive, adaptée à tous les âges et niveaux de compétence numérique.

4.2 Éléments du menu :

- 1. Accueil**
 - Présentation générale de la plateforme.
 - Accès rapide aux jardins populaires ou récemment ajoutés.
 - Mise en avant des fonctionnalités clés et des messages de bienvenue.
- 2. Nos Jardins (Accessible à tous)**
 - Liste des jardins disponibles avec photos et descriptions.
 - Barre de recherche avec filtres avancés (localisation, type de jardin, besoins spécifiques).
 - Carte interactive pour visualiser les jardins à proximité.
- 3. Trouver un jardinier (Pour les propriétaires de jardins connectés)**
 - Liste des amis du vert disponibles dans leur région.
 - Filtres avancés pour rechercher des jardiniers selon leurs compétences (désherbage, plantation, etc.), leur note, et leur disponibilité.
 - Accès aux profils des amis du vert, avec leurs descriptions, évaluations, et heures cumulées.
- 4. Calendrier (Pour les utilisateurs connectés)**
 - Outil interactif pour gérer les créneaux disponibles (propriétaires).
 - Consultation et réservation des créneaux pour les amis du vert.
- 5. Messagerie**
 - Plateforme de communication entre propriétaires et amis du vert.
 - Notifications pour les nouveaux messages.
 - Historique des conversations pour chaque interaction.
- 6. Profil (Pour les utilisateurs connectés)**
 - Informations personnelles (nom, photo, description).
 - Gestion des annonces publiées (propriétaires).
 - Historique des missions passées et heures cumulées (amis du vert).
 - Accès aux évaluations reçues et données statistiques personnelles.
- 7. À propos**
 - Informations sur le projet et ses objectifs.
 - Présentation des valeurs et du fonctionnement de JardinSolidaire.
- 8. Connexion / Inscription (Si non connecté)**
 - Accès aux formulaires d'inscription et de connexion.
 - Options pour s'inscrire en tant que propriétaire ou ami du vert.

9. Déconnexion (*Si connecté*)

- Bouton pour se déconnecter de la plateforme.

4.3 Scénarios utilisateur·ice

1. Propriétaire de jardin : publier une annonce

- Le·la propriétaire de jardin se connecte à son compte.
- Il·elle accède à la section "Mes Jardins" et clique sur "Ajouter un jardin".
- Il·elle remplit un formulaire avec les informations nécessaires (description, photos, type de jardin, besoins spécifiques).
- Il·elle définit les disponibilités sur le calendrier interactif.
- L'annonce est publiée et devient visible pour les amis du vert.

2. Ami·e du vert : rechercher et réserver un jardin

- L'ami·e du vert crée un compte ou se connecte.
- Il·elle accède à la section "Nos Jardins".
- Il·elle utilise les filtres (localisation, type de jardin, besoins, etc.) pour affiner sa recherche.
- Après avoir trouvé un jardin, il·elle consulte la fiche descriptive.
- Il·elle sélectionne un créneau disponible sur le calendrier et clique sur "Réserver".
- Une notification est envoyée au propriétaire pour confirmation.

3. Propriétaire de jardin : rechercher un jardinier

- Le·la propriétaire de jardin se connecte à son compte.
- Il·elle accède à la section "Trouver un jardinier".
- Il·elle utilise les filtres pour rechercher des amis du vert disponibles (compétences, notes, disponibilité).
- Il·elle consulte le profil d'un·e ami·e du vert et clique sur "Contacter".
- Une conversation est initiée via la messagerie intégrée pour organiser les détails.

4. Ami·e du vert : voir et cumuler ses heures

- L'ami·e du vert se connecte à son compte.
- Il·elle accède à la section "Mon Profil".
- Il·elle consulte l'historique des jardins où il·elle a travaillé et le cumul de ses heures.
- Il·elle choisit un jardin qui accepte les réservations d'heures cumulées et planifie une session.

5. Propriétaire ou ami·e du vert : laisser un avis

- Après une interaction (entretien ou visite d'un jardin), l'utilisateur·ice reçoit une notification pour évaluer son expérience.
- Il·elle accède à la section "Avis" et sélectionne l'interaction correspondante.
- Il·elle attribue une note et écrit un commentaire.
- L'avis est publié sur le profil de l'autre utilisateur·ice.

5 . Contraintes techniques et organisationnelles

5.1 Contraintes technique

Les contraintes techniques de JardinSolidaire reflètent notre objectif de concevoir une plateforme fonctionnelle et accessible, tout en tenant compte de nos ressources limitées et de l'absence de technologies imposées.

1. Technologies utilisées

- Langages et outils : Liberté totale dans le choix des langages et frameworks adaptés à nos compétences et besoins. Les choix privilégieront des technologies open-source pour réduire les coûts (exemple : MySQL, JavaScript).
- Autonomie technique : Favoriser des solutions simples et légères pour un développement efficace et rapide.

2. Hébergement

- Utilisation de services d'hébergement gratuits ou à très faible coût, comme 000webhost ou InfinityFree, pour mettre en ligne la plateforme.
- Capacité initiale ciblée : Gestion d'un trafic limité à 50 utilisateurs mensuels actifs dans la phase pilote.

3. Compatibilité multiplateforme

- Responsive Design : La plateforme devra être accessible sur mobile, tablette et ordinateur.
- Tests effectués sur des résolutions standard pour garantir une expérience fluide sur différents supports.
- Compatibilité assurée avec les principaux navigateurs (Google Chrome, Firefox, Safari).

4. Sécurité

- Mise en place d'un système de double authentification par e-mail pour protéger les comptes.
- Utilisation d'un certificat SSL gratuit (ex. : Let's Encrypt) pour sécuriser les échanges de données.
- Cryptage des mots de passe dans la base de données pour éviter toute vulnérabilité.

5. Performances et gestion des données

- Limitation du volume de données pour optimiser l'utilisation des ressources :
 - Taille maximale des fichiers uploadés (images) fixée à 1 Mo.
 - Réduction des éléments lourds pour accélérer le temps de chargement des pages.
- Temps de réponse cible : Chargement des pages inférieur à 3 secondes sur des connexions standard.

6. Budget et ressources

- Budget : 0 €.
- Appui sur des solutions gratuites et open-source pour le développement, la mise en ligne, et les outils tiers.

5.2 Contraintes organisationnelles

1. Délais imposés

- Le projet **JardinSolidaire** doit être finalisé avant le **01/01/2026**, permettant un lancement officiel au début de l'année 2026.
- Cela laisse **49 vendredis** pour travailler sur le projet, soit un total de **7 semaines complètes** en cumulant ces journées.

2. Ressources humaines

- **Équipe actuelle:**
 - **3 conceptrices-développeuses** en formation à AdaTech School, apportant des compétences variées en développement front-end, back-end, et gestion de projet.
- **Compétences disponibles:**
 - Développement web : HTML, CSS, JavaScript, React, Node.js.
 - Gestion de base de données : MySQL.
 - Conception graphique : Utilisation de Figma pour le prototypage et la conception UI/UX.
 - Analyse fonctionnelle : Élaboration de personas, PESTEL, MoSCoW.
- **Limites :** Aucun recours à des prestataires externes, l'intégralité du projet repose sur les compétences internes.

3. Budget alloué

- **Budget initial : 0 €.**
- Solutions exclusivement gratuites ou open-source pour :
 - L'hébergement (services gratuits comme 000webhost ou InfinityFree).
 - Les outils de développement (Visual Studio Code, Figma).
 - La sécurité (Let's Encrypt pour les certificats SSL).

4. Gestion du temps

- Travail organisé autour d'un jour par semaine dédié (vendredi), permettant un suivi régulier de l'avancement.
- Priorisation des tâches à l'aide de la méthode MoSCoW pour respecter les délais avec les ressources disponibles.

6 . Planning et étapes clés (voir tableau)

7 . Ressources nécessaires

1. Ressources humaines

- **Développeuses spécialisées:**
 - **Back-end** : Développement des fonctionnalités serveur, gestion des bases de données, et sécurisation des échanges (ex. : MySQL).
 - **Front-end** : Conception et intégration des interfaces utilisateur responsives et intuitives (HTML, CSS, JavaScript, React).
 - **Full-stack** : Coordination entre les deux aspects pour assurer une fluidité dans l'expérience utilisateur.
- **Designer UX/UI:**
 - Création des maquettes et prototypages (Figma).
 - Conception d'une navigation claire et intuitive, adaptée à tous les supports.
- **Chef de projet:**
 - Organisation et suivi des tâches.
 - Coordination entre les membres de l'équipe et gestion des priorités (ex. : méthode MoSCoW).

2. Ressources techniques

- **Hébergement et infrastructures:**
 - Hébergement web gratuit ou peu coûteux (ex. : 000webhost, InfinityFree).
 - Base de données MySQL pour le stockage des utilisateurs, annonces, et réservations.
 - Certificat SSL gratuit (Let's Encrypt) pour sécuriser les échanges.
- **Outils de développement:**
 - IDE : Visual Studio Code pour l'écriture et la gestion du code.
 - Outils de gestion de version : Git et GitHub pour le suivi des modifications et la collaboration.
 - Bibliothèques et frameworks : Utilisation de technologies open-source selon les besoins identifiés (ex. : jQuery, Bootstrap, etc.).

3. Ressources matérielles et financières

- **Budget global:**
 - **Initial : 0 €.** Le projet repose sur des outils gratuits et open-source.
- **Matériels:**
 - Ordinateurs personnels des membres de l'équipe.
 - Connexion Internet stable pour le développement, les tests, et la mise en ligne.
- **Logiciels et abonnements:**
 - Outils gratuits : Figma pour le design, GitHub pour la gestion des versions, et hébergeurs gratuits.
 - Aucun abonnement payant requis pour la phase initiale du projet.

8. Sécurité et Accessibilité

8.1 . Sécurité

La sécurité des données et des échanges est une priorité pour le projet **JardinSolidaire**. Les mesures mises en place incluent :

- **Protection des données utilisateur :**
 - Conformité au **RGPD** (Règlement Général sur la Protection des Données) pour garantir la confidentialité des informations collectées.
 - Limitation des données collectées au strict nécessaire (nom, e-mail, localisation générale).
 - Transparence dans l'utilisation des données avec des consentements explicites lors de l'inscription.
- **Sécurisation des échanges :**
 - Mise en place d'un certificat SSL (Let's Encrypt) pour sécuriser les communications sur la plateforme (HTTPS).
 - Cryptage des mots de passe dans la base de données (utilisation de fonctions telles que password_hash()).
- **Prévention des attaques :**
 - Protection contre les attaques courantes comme les injections SQL et les attaques XSS.
 - Limitation des tentatives de connexion pour prévenir les attaques par force brute.

8.2 . Accessibilité

JardinSolidaire vise à être inclusif et accessible à tous les utilisateurs, y compris les personnes en situation de handicap, en suivant les normes du **RGAA** (Référentiel Général d'Amélioration de l'Accessibilité).

- **Conception accessible :**
 - Respect des bonnes pratiques de conception web : navigation clavier, contraste des couleurs, et textes alternatifs pour les images.
 - Utilisation de balises HTML sémantiques pour une navigation simplifiée avec des technologies d'assistance (lecteurs d'écran).
- **Tests d'accessibilité :**
 - Réalisation de tests réguliers pour garantir que le site est conforme aux standards d'accessibilité (WCAG 2.1 niveau AA).
 - Adaptation des fonctionnalités pour différents handicaps (moteurs, visuels, auditifs).
- **Responsive Design :**
 - Compatibilité avec les différents appareils (mobile, tablette, ordinateur) pour élargir l'accès à un maximum d'utilisateurs.