**Paul BONHOMME**

**Meriem HANSALI**

# Rapport de projet

# Jardin d'intérieur connecté

Projet réalisé du 30 septembre 2022 au 3 mars 2023

**Tuteurs :**

Michel CHEMINAT

Patrice LAURENCOT

**Institut Supérieur d'Informatique, de Modélisation et de leurs Applications**

**Université Clermont Auvergne – Campus des Cézeaux**

**63178 AUBIERE**

# Remerciements

Nous tenons à remercier notre tuteur référent M. Michel CHEMINAT ainsi que notre référent école M. Patrice LAURENCOT. De plus, nous tenons à remercier M. Christophe TILMANT. En effet, ils nous ont accompagné tout au long de cette période de projet tuteuré. Ils nous ont été d'une grande aide durant son développement.

# Table des matières



N.B : les mots suivis d'un astérisque (*) sont définis dans le lexique

# Table des figures

# I. Introduction

Nous sommes actuellement en deuxième année d'école d'ingénieur à l'ISIMA, Institut Supérieur d'Informatique, de Modélisation et de leurs Applications. Au début de cette seconde année, nous avons découvert le thème qui allait orienter nos projets pour les 5 prochains mois appelé « Easy Data ».

Nous avions deux possibilités, choisir de travailler sur les projets en lien avec les autres écoles d'ingénieur de la ville telles que Sigma et Polytech ou proposer notre propre sujet. Nous avons décidé d'en proposer un car nous étions inspirés et toute l'équipe s'est mise d'accord sur le sujet de notre futur projet.

Nous devions choisir un projet qui nous permettrait de traiter des données, de manière sécurisé et facile. Nous voulions quelque chose qui allait pousser notre réflexion ainsi que notre sens de la conception et du développement.

Nous avons choisi de créer un jardin autonome connecté qui nous permettrait de faire pousser des plantes ou des herbes aromatiques telles que le basilic, la menthe ou du romarin. Avec le jardin vient une application Web nommé « iGardenConnect » qui permet de suivre l'état de son jardin au fil du temps. En effet, la possibilité de créer son compte de manière sécurisée, ajouter son jardin, et consulter les données des capteurs permet à l'utilisateur d'être régulièrement à jour sur l'état de ses plantes. Pour faire le lien entre l'application web, les capteurs du jardin connecté et la base de données, nous avons décidé de créer une API* [1] dont nous détaillerons les fonctionnalités par la suite.

Durant ces cinq mois, nous avons été amenés à utiliser différentes technologies, de la conception de nos applications, du choix de leurs fonctionnalités à leurs développements. Nous avons, chacun, élargis nos compétences dans ce domaine.

Comment permettre à un utilisateur de cultiver toute l'année, sans effort, directement dans son lieu de vie qu'il soit en ville ou à la campagne ?

Afin de présenter notre projet, nous aborderons dans un premier temps le contexte du projet. Ensuite, nous verrons les outils utilisés et parlerons de la conception préalable à la réalisation et nous terminerons en détaillant le développement du projet en lui-même. Enfin nous dresserons un bilan technique et personnel.

# II. Contexte du projet

## A. Définition du besoin

La culture des plantes est une activité courante dans le monde, pratiquée à la fois à des fins professionnelles et de loisirs. Cependant, les cultivateurs rencontrent souvent des difficultés liées à la croissance de leurs plantes. Pour résoudre ce problème, nous avons créé un jardin d'intérieur connecté, qui permet de surveiller et de contrôler facilement les paramètres vitaux des plantes pour assurer leur croissance optimale.

Les trois paramètres les plus importants pour assurer le succès de notre concept sont la température, l'eau et la lumière. La température joue un rôle crucial dans la croissance des plantes. En effet, une température trop élevée ou trop basse peut entraîner la destruction ou le ralentissement des plantes. Cela peut être particulièrement préjudiciable aux plantes d'intérieur qui n'ont pas la possibilité de s'adapter aux changements de température de l'environnement extérieur. Avec notre jardin connecté, les utilisateurs peuvent facilement réguler la température pour assurer des conditions optimales pour leurs plantes.

De même, l'eau est essentielle à la survie des plantes. Les plantes libèrent de l'humidité lorsqu'elles respirent, ce qui rend l'air ambiant plus humide. Cela signifie qu'un arrosage régulier est nécessaire pour compenser cette perte d'eau. En effet, un manque d'eau peut avoir des effets néfastes sur la croissance des plantes. À l'inverse, un arrosage excessif peut noyer les racines des plantes. Notre jardin connecté permet aux utilisateurs de surveiller de manière précise la quantité d'eau dont leurs plantes ont besoin, assurant ainsi leur croissance optimale.

La lumière est indispensable à la photosynthèse, un processus qui permet aux plantes de produire de la nourriture en utilisant la lumière, l'eau et le dioxyde de carbone. Cependant, il est important de savoir que chaque plante a des besoins spécifiques en termes de lumière. Certaines plantes ont besoin de beaucoup de lumière directe du soleil, tandis que d'autres peuvent pousser à l'ombre ou avec une lumière indirecte

Chaque plante a des besoins spécifiques en termes d'humidité, de lumière et de température, qui doivent être pris en compte pour assurer leur croissance optimale. Ces trois paramètres sont donc accessibles directement depuis notre jardin connecté, grâce à des outils conçus à cet effet. Les utilisateurs peuvent ainsi contrôler la température, la lumière et surveiller l'eau de manière efficace et intuitive.

## B. Objectif du projet

Le projet de jardin d'intérieur connecté vise à concevoir un système qui permet de cultiver des plantes à l'intérieur de la maison en utilisant une technologie connectée. L'objectif principal est de faciliter la culture des plantes en fournissant une solution automatisée qui permet de réguler leur croissance, leur alimentation et leur hydratation, tout en contrôlant les paramètres environnementaux tels que la température, l'humidité et la lumière.

Le système sera basé sur une architecture de microservices*, avec une API* en C# .NET Core pour la gestion des données et une application front-end* en Next JS pour l'interface utilisateur. Le front-end offrira une interface graphique agréable et intuitive pour faciliter l'utilisation du système. Les utilisateurs pourront ainsi visualiser les données de leurs jardins, ajuster les paramètres de croissance, choisir la plante à élever, et recevoir des notifications en cas d'alertes ou de problèmes avec leurs plantes.

L'API en C# .NET Core sera responsable de la gestion des données du système, notamment de la collecte et du stockage des données sur les plantes, des paramètres de croissance et des paramètres environnementaux. Elle permettra également aux utilisateurs d'interagir avec le système en fournissant des fonctionnalités pour la création, la lecture, la mise à jour et la suppression de données.

Le projet de jardin d'intérieur connecté offre une solution pratique pour ceux qui souhaitent cultiver des plantes à l'intérieur de leur maison. L'objectif est de rendre la culture des plantes plus facile, plus efficace et plus agréable pour les utilisateurs en leur offrant une expérience connectée et automatisée.

## C. Technologies et outils utilisés

Nous avons pris le temps d'analyser les différents les technologies et outils disponibles afin de choisir ce qu'il convient le mieux à notre projet. Pour cela, nous avons pris en compte plusieurs critères :

1- **Adéquation avec les besoins fonctionnels et techniques** : c'est-à-dire des technologies qui nous permettent d'une part la création d'une API qui gère des données de jardins et de leurs capteurs et d'autre part la réalisation d'une interface graphique pour les utilisateurs.

2- **La facilité de développement et la maintenance :** nous devions développer ce projet et le livrer en un temps imparti ; il est important de choisir des technologies qui facilitent le développement et la maintenance du projet, pour les développeurs, une documentation claire et des outils de développement efficaces.

3- **Le coût :** nous avons aussi pris en compte le coût des outils tels que les frais de licence et frais de support.

4- **La sécurité :** c'est un critère important lors du choix d'une technologie. Il est important de s'assurer que la technologie choisie est capable de fournir un niveau de sécurité adéquat pour protéger les données de l'utilisateur et l'infrastructure du projet.

À la suite de la phase de recherche et d'analyse, notre choix s'est porté sur ces langages et outils :

- **Langage C# :** Langage de programmation moderne, élégant et fortement typé qui est utilisé pour créer des API en C# .NET. Il offre une grande flexibilité et permet de créer des applications robustes et évolutives.

- **ASP.NET Core :** Framework* web open-source* qui fournit des bibliothèques et des outils pour faciliter la création de services web. Il est multiplateforme et peut être déployé sur Windows, Linux et macOS.

- **Entity Framework Core :** Framework de mapping* objet-relationnel (ORM) qui permet de mapper les données entre une base de données et un modèle de domaine. Nous l'avons utilisé pour créer notre modèle c'est-à-dire nos classes principales à partir de la base de données que nous avons implémentée.

- **Swagger** : Framework open source qui permet de décrire, de produire et de consommer des API RESTful. Il fournit des outils pour la documentation, la génération de code client et la validation de l'API.

- **Visual Studio 2022 :** Visual Studio est un environnement de développement intégré (IDE) utilisé pour créer des applications .NET. Il est également utilisé pour créer des API en C# .NET. Visual Studio fournit des fonctionnalités telles que le débogage*, la mise en forme de code et la génération de code pour faciliter le développement d'API.

- **Microsoft SQL Server :** Système de gestion de base de données fiable et performant qui utilise le langage SQL. Il nous a permis de créer toutes les tables SQL*, de modéliser les relations entre elles ainsi que de visualiser les données liées aux jardins.

- **NextJS** : Framework de développement web open-source basé sur React*. Il permet de créer des applications web rapides et performantes. Un des plus gros avantages qui nous a poussé à choisir NextJS est sa gestion avancée des routes, qui facilite la création de chemins d'accès pour les pages de l'application. Il permet également de gérer facilement les requêtes HTTP* vers des APIs.

- **StarUML :** Logiciel utilisé pour faire des diagrammes de modélisation UML*.

- **GanttProject :** Logiciel utilisé pour réaliser les diagrammes de Gantt.

- **Git/Gitlab*** : Notre mode de travail a été de créer une branche* GIT par fonctionnalité ou par type de fonctionnalité. Par exemple : pour afficher, créer, modifier et supprimer les données liées à une plante nous avons créé la branche « crudPlant ». Un autre exemple ; la fonctionnalité « création de compte d'un nouvel utilisateur » a été développée sur la branche correspondante appelée « authUser ».

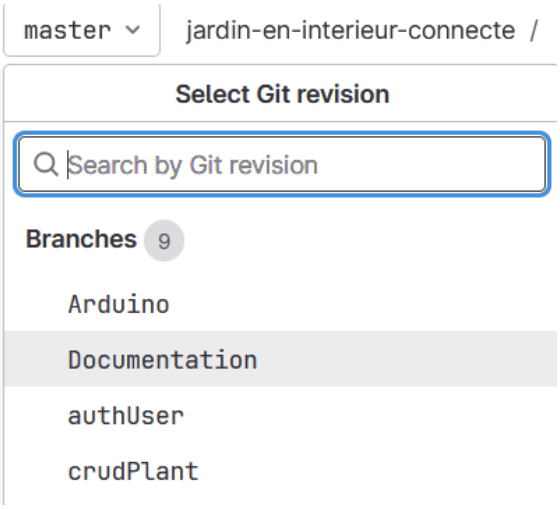



*Figure 1 - Capture d'écran de quelques branches utilisées*

# III. Organisation prévisionnelle

Après avoir découvert les outils et les technologies, nous avons défini un planning détaillé à partir de tous les objectifs détaillant les tâches principales à réaliser durant ce projet. Pour cela, nous avons élaboré un Gantt prévisionnel pour mieux nous organiser et avoir un meilleur aperçu de l'évolutivité du projet.

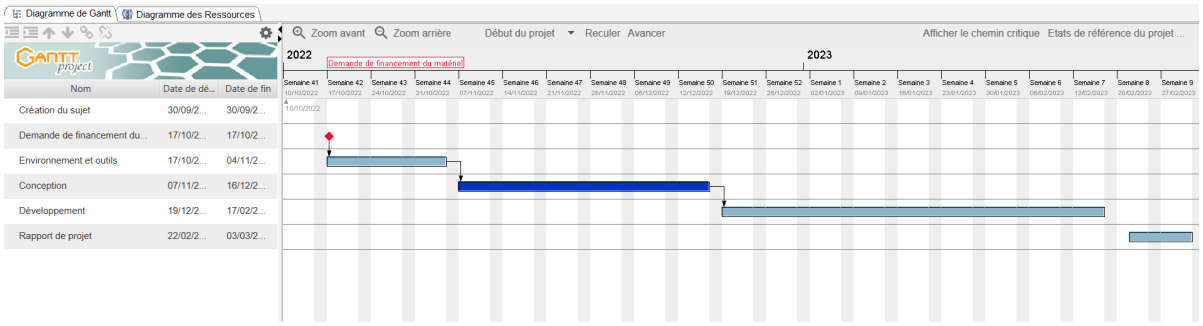



*Figure 2 – Diagramme de Gantt prévisionnel*

Nous avons décidé de découper le projet en 4 grandes parties. La première consistait en l'élaboration de notre sujet de projet, ainsi que la demande de financement du matériel dont nous allions avoir besoin. Cette première partie s'est déroulée avec les quatre membres du groupe. Par la suite, nous avons divisé le projet et défini les binômes.

La deuxième partie a été la mise en place de l'environnement sur lequel nous allions travailler et l'installation sur nos ordinateurs personnels de tous les outils et logiciels cités précédemment. Après avoir maitrisé notre futur environnement de travail, nous avons commencé à créer des sketchs des futures pages Web ainsi que tous les diagrammes nécessaires à la conception du projet.
Ensuite, la dernière partie consistait en le développement du projet avec la création de l'API, et de l'interface Web.

Sur la figure 3, nous observons le diagramme de Gantt réel représentant l'avancement réel des tâches. Par exemple, nous avons commencé le développement de l'API le 04/01/2023. Sur le diagramme de Gantt prévisionnel il était prévu que cette tâche débute le lundi 19/12/2023 directement après que la tâche « Conception de la base de données » soit terminée.

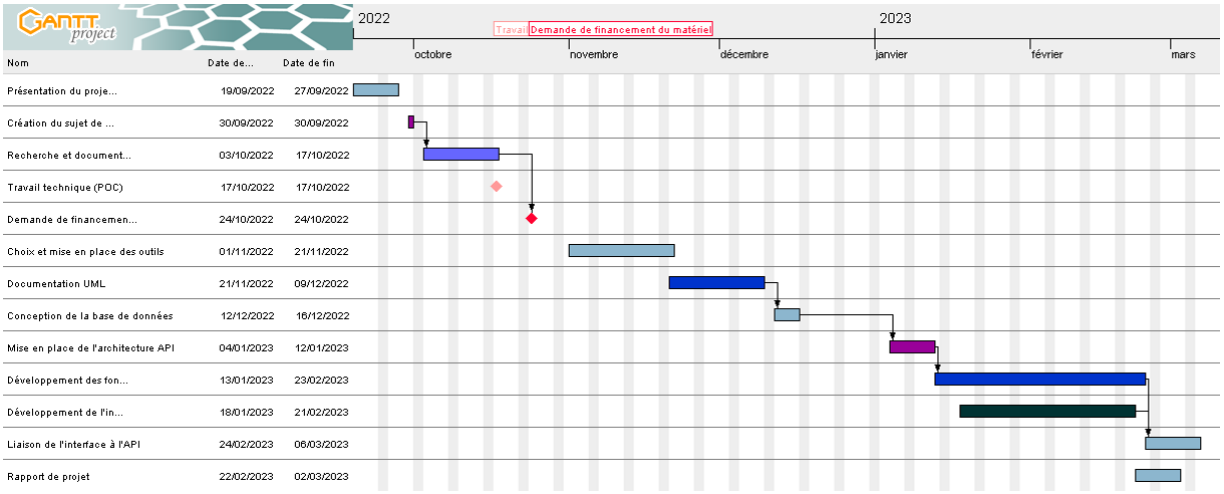



*Figure 3 - Diagramme de Gantt réel*

Cet écart entre la planification initiale et réelle est expliqué par les vacances de Noël car nous n'avons pas pu travailler sur le projet durant cette période. Ce retard s'est aperçu sur la suite des tâches réalisées.

Aussi, nous voyons que la date 24/10/2022 représente une date **jalon*** car la tâche « Demande de financement du matériel » aperçue en rouge et définie par un losange est une tâche **critique***.

En effet, la préparation de la liste de matériels et le dépôt de la demande de financement avant les vacances de la Toussaint a été une étape clé à la progression du projet. Sans le matériel, nous n'aurions pas pu assurer sa réalisation.

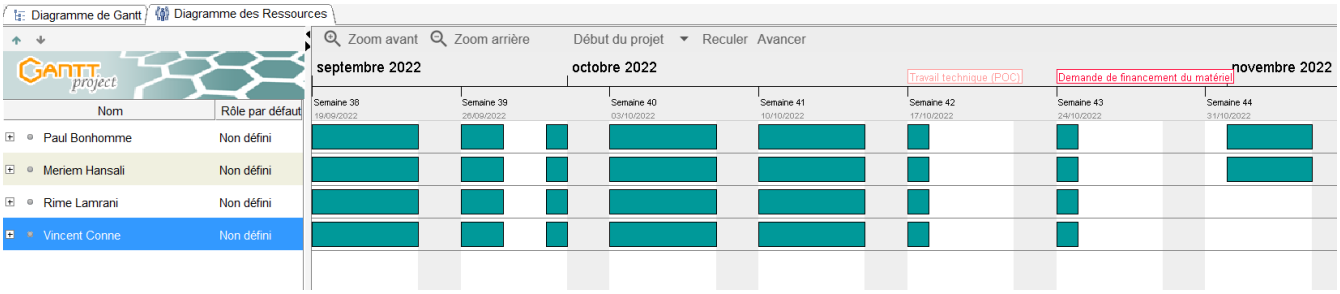



*Figure 4 - Diagramme de ressources*

Un diagramme de ressources est fourni par le logiciel (figure 4) ; le nombre de rectangles verts sur chaque colonne représente le nombre de personnes qui ont travaillé sur la tâche. Par exemple, le 24/10/2022 nous voyons que la tâche « Travail technique (POC) » a été réalisée par les quatre membres du projet.

Après avoir structuré le déroulement des futures semaines, nous entrions dans la partie conception du projet.

# IV. Conception

Avant de commencer le développement de l'application, une phase de conception a été réalisée afin d'identifier les besoins métiers des futurs utilisateurs, de comprendre les différents types de données à manipuler, d'implémenter la base de données et de définir les futures vues de l'interface.

## A. Diagramme de cas d'utilisations :

Un diagramme de cas d'utilisation est utilisé pour donner une vision globale du comportement fonctionnel de notre système logiciel. Il permet de présenter les acteurs, les différentes actions métiers de l'interface et les dépendances entre chacune.

Réaliser un diagramme en amont du développement du projet nous a aussi permis de définir les limites métiers de notre application. Nous avons décrit les scénarios nominaux et alternatifs des différents cas d'utilisation (**voir figures 34, 35, 36 dans les annexes).**

Nous avons utilisé le logiciel StarUML pour réaliser ce diagramme.

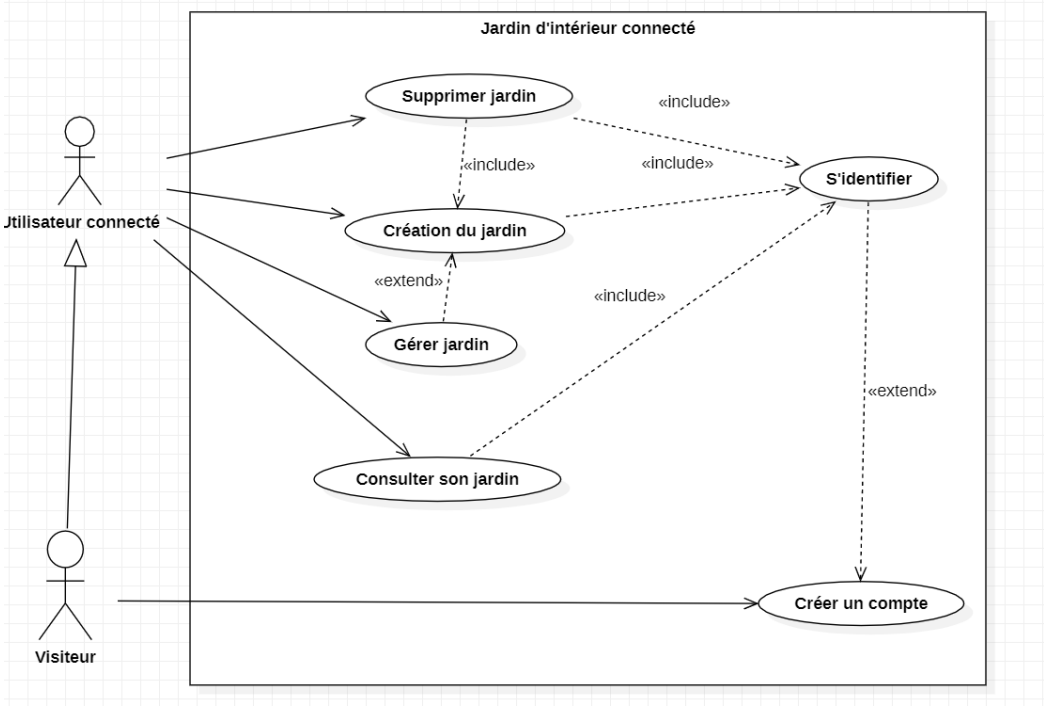



*Figure 5 - Diagramme de cas d'utilisations de iGardenConnect*

## B. Définition des vues de l'interface :

Nous avons réalisé plusieurs sketchs afin de visualiser nos idées et de définir concrètement le rendu final de l'interface de notre application Web. Faire des sketchs nous ont aussi permis en tant que binôme de se mettre d'accord sur le placement et la forme des éléments que nous avons choisis.

Nous avons utilisé le site « moqups.com » pour réaliser ces différentes vues.

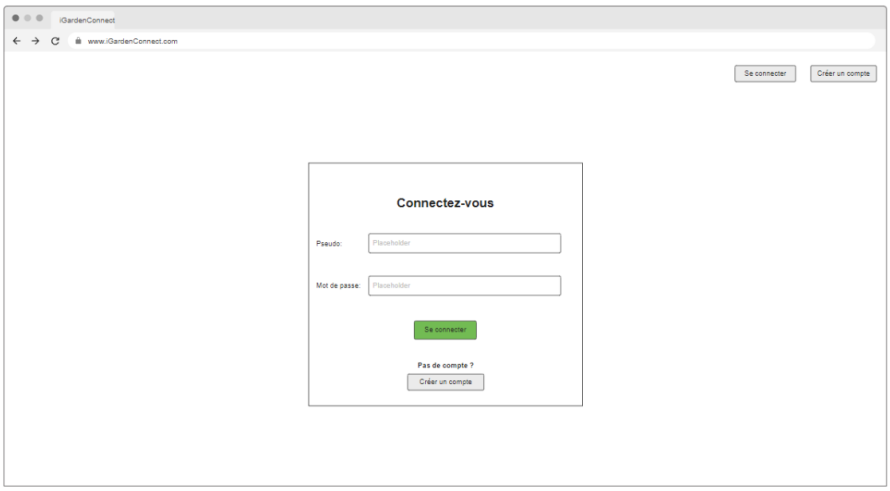



*Figure 6 - Sketch qui illustre la page de connexion d'un utilisateur non connecté*

L'objectif n'étant pas de faire des visuels très poussés, nous nous sommes concentrés sur le fonctionnel.

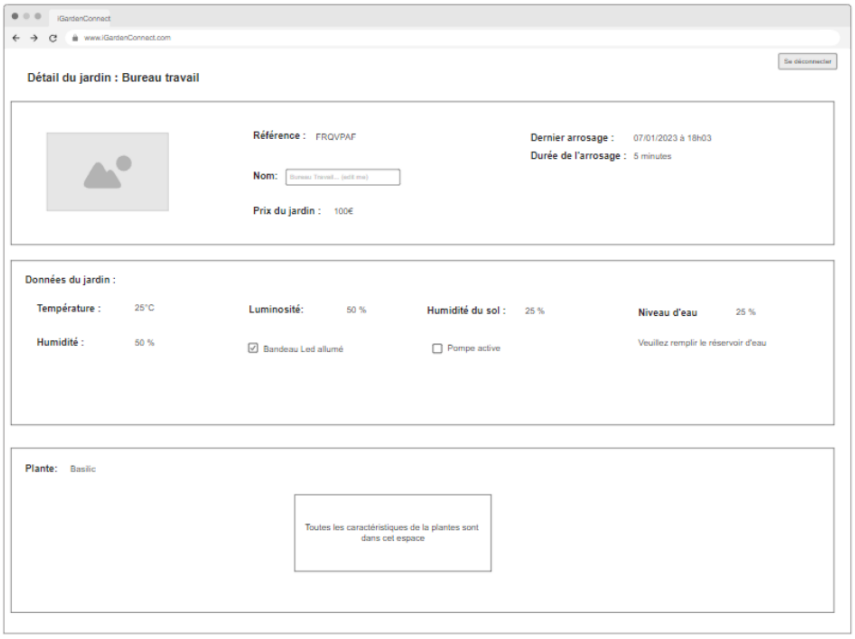



*Figure 7 - Sketch qui illustre la page de détail d'un jardin*

# C. Modèle de données

## 1. Conception de la base de données :

Dans le but de sauvegarder les données des utilisateurs tels que leurs comptes créés ainsi que leurs jardins ; nous avons commencé par réfléchir à notre modèle conceptuel de données (figure 8) dans le but d'avoir une bonne organisation de nos données via des entités ; détailler leurs identifiants, ainsi que désigner les attributs utilisés pour chaque entité.

Par exemple : nous avons eu besoin d'une table « Sensors » qui recense tous les capteurs prédéfinis avec leurs noms, leurs marques, leurs prix.

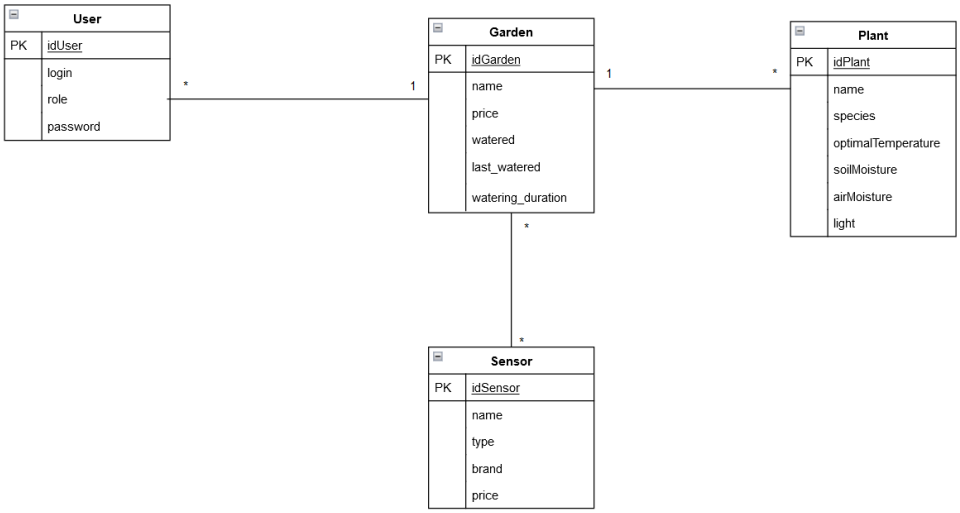



*Figure 8 - Modèle conceptuel de données iGardenConnectDB*

Nous avons par la suite réalisé le modèle relationnel qui a permis de représenter les relations entre nos entités en détaillant leurs clés primaires* et étrangères*. Ce modèle est illustré sur la figure ci-dessous.

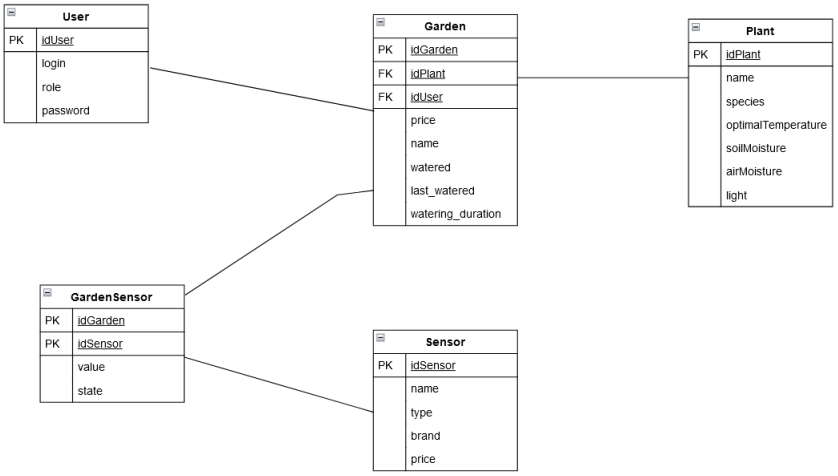



*Figure 9 - Modèle relationnel de données iGardenConnectDB*

## 2. Création de la base de données :

Une fois la conception de notre base de données finalisée, l'étape suivante a été de la mettre en place de manière concrète. Nous avons choisi Microsoft SQL Server comme système de gestion de base de données car d'une part, il est parfaitement compatible avec le langage SQL que nous utilisons. D'autre part, Microsoft SQL Server est connu pour sa grande performance en matière de traitement des transactions en temps réel.

Nous avons donc créé nos tables SQL sur ce logiciel qui nous a facilité la gestion des contraintes de tables et des références de clés étrangères entre elles.

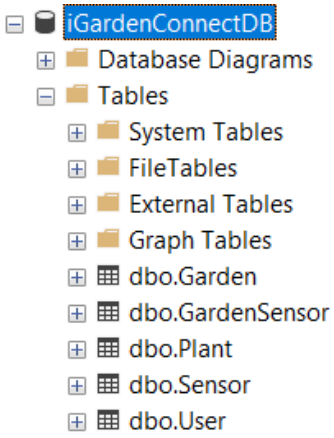



*Figure 10 - Base de données iGardenConnectDB sur Microsoft SQL Server 2018*

Une fois notre modèle de données mis en place en local sur nos ordinateurs (figure 10), nous avons entamé la phase de développement de notre application.

# V. Développement

## A. Description de la structure de iGardenConnectAPI

La plupart des applications ASP.NET Core sont basées sur le même principe. Elles possèdent au moins un contrôleur, un service, un dépôt et un modèle.

Dans l'exemple qui va suivre nous utiliserons les plantes pour illustrer le fonctionnement de l'application.

Chaque fonctionnalité respecte une règle de nommage qui permet de les différencier. Par exemple « api/Plant » permet de récupérer les plantes existantes dans la base de données. Son contrôleur aura le nom « PlantController », son service « PlantService » et son dépôt « PlantRepository ».

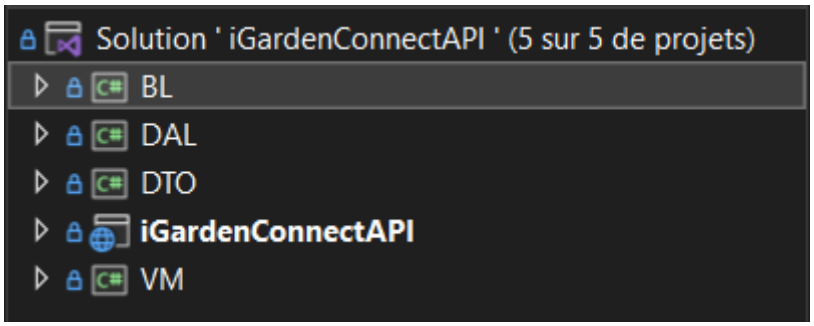



*Figure 11 - API iGardenConnect sur Visual Studio 2022*

Chaque application est composée de plusieurs projets ayant chacun une action et un rôle unique. Il y a donc le projet principal comportant les contrôleurs, un projet BL pour « Business Layer » faisant le lien entre les contrôleurs et les dépôts, un projet DAL pour « Data Access Layer » permettant la récupération des données, et enfin un projet DTO pour « Data Transfer Object » contenant le modèle qui sera utilisé dans toute l'application. Les différents projets sont illustrés sur la figure 11 ci-dessus.

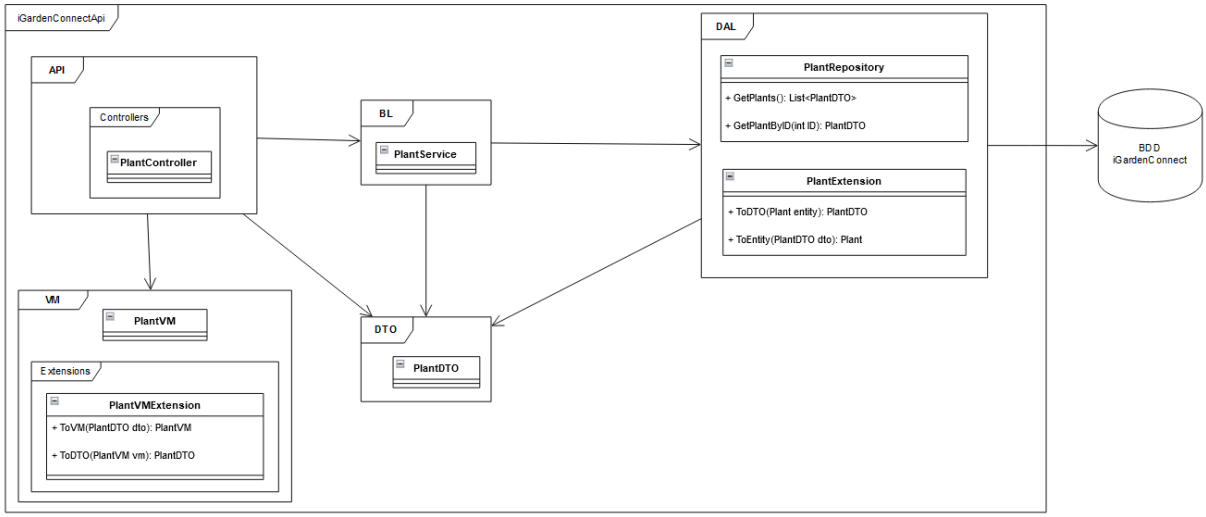



*Figure 12 - Diagramme simplifié de la structure des applications avec l'exemple d'une Plante*

Notre application Web et Swagger possèdent une interface utilisateur et affichent des données. Ces données sont de type « View-Model » qui est un autre modèle défini dans un projet VM. C'est ce modèle-là qui servira à l'affichage.

Nous pouvons observer sur la figure 12 ci-dessus un schéma simplifié sur la structure et les dépendances entre chaque projet de l'API. Aussi, nous observons des objets de type PlantDTO et PlantVM. Ces objets sont différenciés car il se peut que certains attributs ne servent qu'à effectuer des opérations et ne soient pas utiles lors de l'affichage des objets.

Les APIs utilisent en plus l'outil Swagger* afin d'avoir une interface graphique générée automatiquement et permettant d'accéder à tous les contrôleurs et à toutes leurs méthodes* très simplement. Voici un exemple d'une API utilisée avec Swagger :

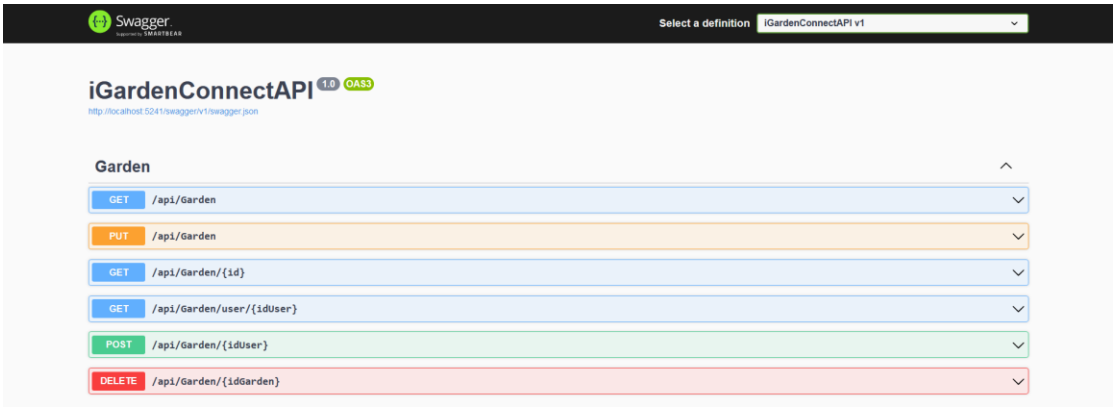



*Figure 13 - Outil Swagger pour une application ASP .NET*

## 1. Création du modèle :

L'approche de développement que nous avons utilisé s'appelle "Database First". Elle consiste à créer notre schéma de base de données en premier (expliqué dans la partie « Modèle de données ») et par la suite l'utiliser pour générer notre modèle (figure 14) au niveau de l'API. Il existe aussi l'approche « Code First » qui consiste à écrire le modèle en premier au travers de classes et de générer la base de données par la suite.

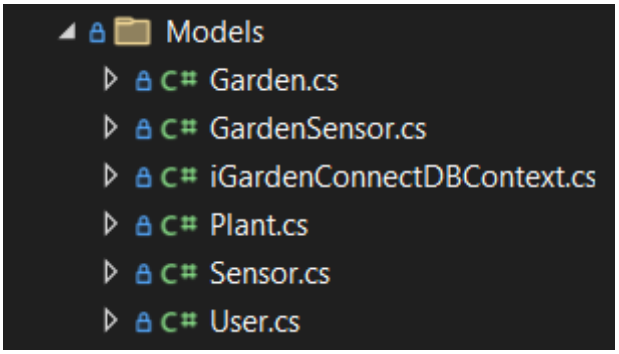



*Figure 14 - Classes d'entités*

Dans cette approche, nous avons utilisé l'outil « EF Core Power Tools » * qui nous a permis de générer rapidement nos classes d'entités à partir du schéma de base de données. Nous avons illustré notre modèle avec un diagramme de classe UML **(voir Figure 37 dans les annexes)**. Cet outil représente un gain de temps important lors de notre phase d'écriture de code.

Une fois notre modèle crée, nous avons commencé à implémenter les fonctionnalités de notre projet en respectant l'architecture de base d'une API .Net Core.

## 2. Les contrôleurs

Un contrôleur est le point de départ d'une fonctionnalité. La figure 15 affiche tous les contrôleurs et si nous cliquons dessus cela va afficher toutes les actions qu'il est possible de réaliser.

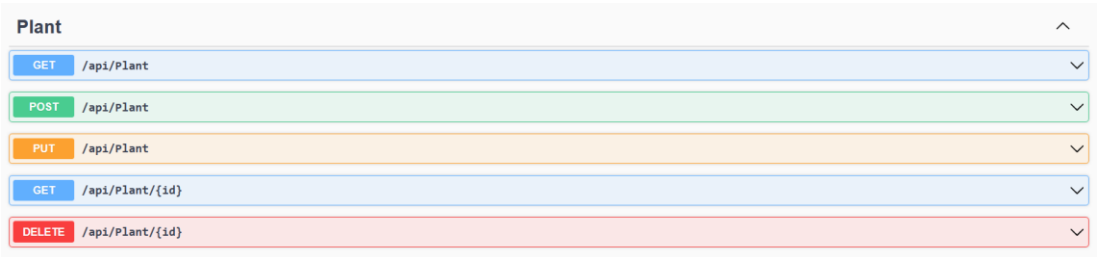



*Figure 15 - Liste des méthodes dans le PlantController*

Nous pouvons voir ci-dessus la liste des méthodes disponibles dans le **PlantController**. Afin d'accéder à la méthode en question, une requête HTTP* est envoyé à une URL*. Ces URLs sont des points de terminaison, en anglais appelé « Endpoint ». Nous pouvons observer différents endpoints de l'API sur la figure 15, à droite des mots clés « GET », « POST », « PUT », « DELETE ».

L'URL écrite dans l'encadrée est définie au-dessus de la méthode en question comme nous pouvons le voir sur la figure ci-dessous :

```
/// <summary>
/// Get all the plants
/// </summary>
/// <returns>List of plants</returns>
#region GET
[HttpGet]
[Route("")]
0 références
public IEnumerable<PlantVM> Get()
{
    var plantsDTO = _plantService.Get();

    return plantsDTO.Select(dto => dto.ToVM());
}
```

*Figure 16 - Exemple de route pour une méthode de type GET*

L'URL ou point de terminaison sont définies en utilisant la balise Route juste avant le chemin. De plus, la méthode HTTP est définie au-dessus de la route grâce au mot-clé « HttpGet ». Cette méthode définit le fait que nous allons récupérer des données sans les modifier. Il existe d'autres mots-clés définissant des méthodes HTTP comme « HttpPost » qui signifie que nous allons ajouter des données. « HttpPut » correspond à la mise à jour de données qui sont déjà présentes en base de données ou à l'ajout de toutes nouvelles données. Enfin, « HttpDelete » signifie que nous allons supprimer des données de la base de données.

Nous avons vu précédemment que « PlantController » possédait plusieurs méthodes de type GET. La première récupère toutes les plantes présentes dans la base de données iGardenConnectDB. La deuxième récupère une plante en fonction de son identifiant idPlant, elle est illustrée sur la figure 17.

```
/// <summary>
/// Get a plant from its reference
/// </summary>
/// <param name="id">Reference of the plant</param>
/// <returns></returns>
[HttpGet]
[Route("{id}")]
1 référence
public PlantVM Get(int id)
```

*Figure 17 - Exemple de route pour une méthode de type GET avec un paramètre*

Le paramètre requis est défini dans l'URL entre accolades ; dans l'exemple ci-dessus nous avons besoin de l'identifiant de plante.

Si l'utilisateur fait une requête à l'API avec pour URL : « http://localhost:5241/api/Plant/3 », l'API trouvera directement la méthode appelée GET. Elle appellera la classe PlantService pour récupérer les données associées et fera des vérifications. En fonction de celle-ci, elle renverra une erreur ou les données sous le format que nous avons détaillé précédemment grâce à la méthode d'extension « ToVM() ». Une méthode d'extension permet d'ajouter des fonctionnalités à des objets sans avoir à modifier le code source, dans ce cas à un PlantDTO de devenir un PlantVM. Ces méthodes permettent aussi d'éviter la duplication de code.

## 3. Les Services

Un service est un lien entre les données récupérées dans le « repository » et le contrôleur. C'est une étape qui permet si nous le souhaitons de modifier les objets.

```
#region GET
/// <summary>
/// Get all plants
/// </summary>
/// <returns>List of all plants</returns>
2 références
public IEnumerable<PlantDTO> Get()
{
    return _plantRepository.Get();
}
```

*Figure 18 - Exemple méthode GET du PlantService*

Dans notre exemple avec les plantes, nous n'avons pas besoin de modifier les données. Nous les faisons simplement remonter.

## 4. Les dépôts

Le dépôt que nous appelons « Repository » est la classe qui va récupérer les données. À l'aide de requêtes HTTP, il pourra faire un appel à la base de données et y effectuera des requêtes SQL.

```
#region GET
/// <summary>
/// Returns all the plants in the database
/// </summary>
/// <returns>IEnumerable<PlantDTO></returns>
- références
public IEnumerable<PlantDTO> Get()
{

    return _dbcontext.Plants.ToList().Select(p => p.ToDTO()).ToList();
}
```

*Figure 19 - Exemple méthode GET du PlantRepository*

Ci-dessus l'exemple de la méthode « GET » appelée par le Service. La variable « _dbcontext » correspond à la base de données iGardenConnectDB. La requête va donc aller chercher dans la table « Plant » le premier objet qui remplira les critères voulus. Dans notre exemple plus tôt, la première plante ayant pour identifiant idPlant égal à 3.

Nous récupérons des données sous format SQL, ces données sont transformées en objet C# grâce au framework « Entity Framework ». Ensuite nous les transformons en GardenDTO qui est le type d'objet utilisé dans l'application grâce à la méthode d'extension « ToDTO() ».

Nous avons réalisé le schéma ci-dessous pour simplifier le comportement de notre API à la suite d'une requête GET avec des paramètres.

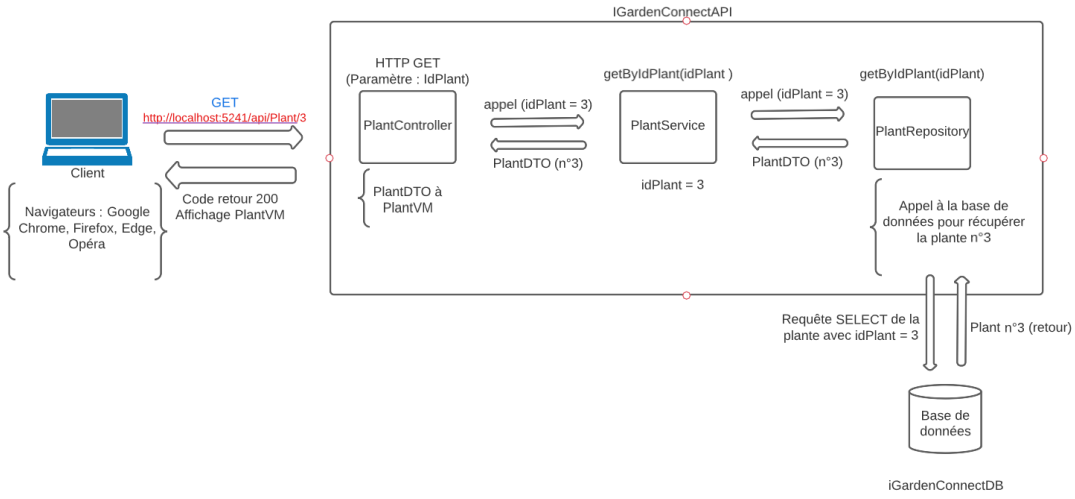



*Figure 20 - Schéma expliquant la récupération de la plante numéro 3*

Entity Framework Core permet de créer une couche d'accès aux données liées à une base de données relationnelles. C'est une solution pour laquelle Microsoft a opté afin que les développeurs se concentrent plus sur la création des fonctionnalités métiers de leurs applications plutôt que de devoir passer du temps à écrire des requêtes SQL.

Entity Framework permet les quatre types d'opérations représentées par l'acronyme CRUD :

- Create : création d'une entité.
- Read : lecture d'entité(s).
- Update : mise à jour d'une entité.
- Delete : suppression d'une entité.

Nous avons vu précédemment, l'utilisation de la méthode GET pour la récupération de la plante avec l'identifiant n°3, voici un exemple avec la méthode POST utilisé lors de l'ajout d'une entité plante.

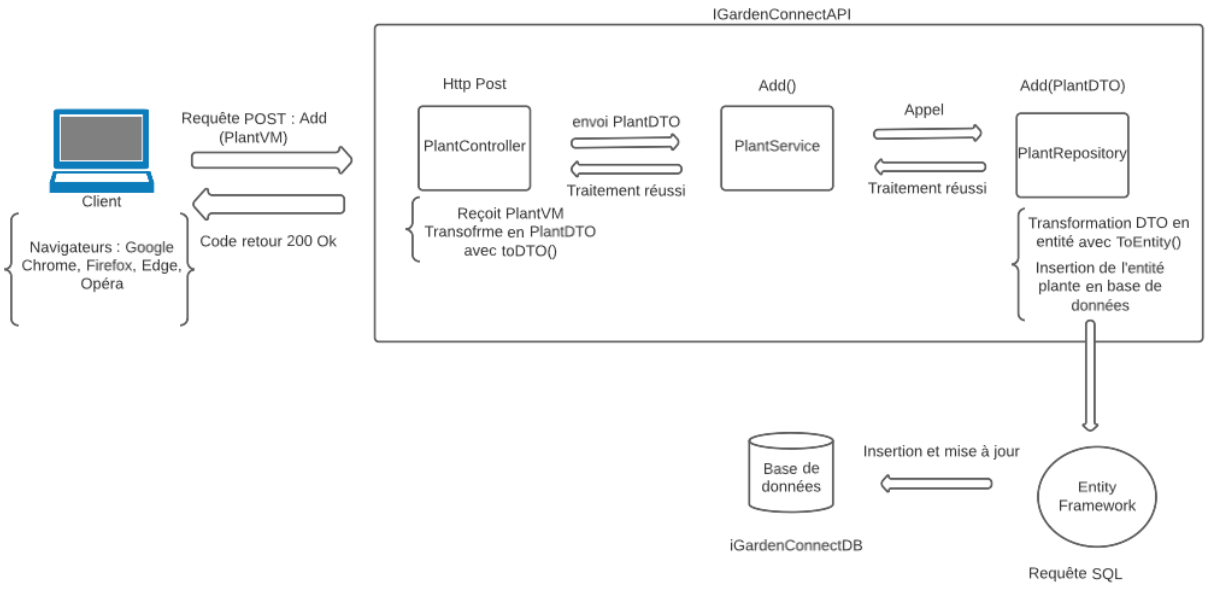



*Figure 21 - Schéma d'une requête POST d'ajout de plante*

La figure 21 ci-dessus représente un schéma clair sur le fonctionnement de notre système quant à l'ajout d'une nouvelle entité « Plant » en base de données. Ce même schéma de fonctionnement est utilisé pour la réalisation du CRUD de toutes nos entités.

L'API iGardenConnectAPI n'est pas seulement utilisée par l'utilisateur au niveau de l'interface graphique mais aussi parallèlement par le jardin grâce à sa carte Arduino (Figure 22). En effet, des requêtes sont envoyées à l'API afin de mettre à jour en continu des données telles que les valeurs réelles des capteurs, le statut et la durée d'arrosage des jardins. Pour cela, des endpoints ont été développées au niveau de l'API et sont spécifiques à la mise à jour de données effectuée par le jardin.

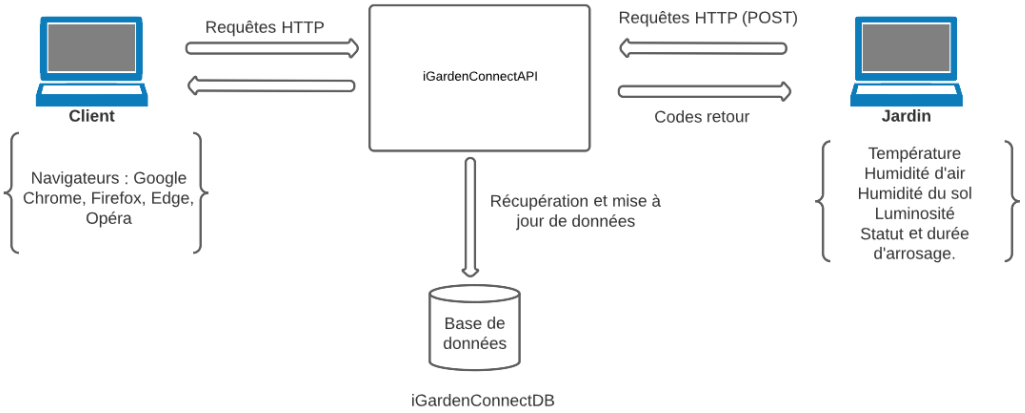



*Figure 22 - Utilisation de l'API par l'interface graphique et le jardin*

C'est ainsi que nous avons principalement fonctionné pendant le projet ; nous avons dû réaliser en premier lieu le CRUD des différentes entités (Jardin, Plante, User, Sensor..). Après avoir réalisé différents tests sur les différentes opérations grâce à l'outil Swagger, nous sommes passés au développement de la partie front, c'est-à-dire l'interface utilisateur.

## B. L'interface utilisateur

### 1. Présentation de Next.js

L'interface utilisateur est un élément clé de notre jardin connecté. Elle permet à l'utilisateur de visualiser et d'interagir avec les données collectées par les capteurs et les actionneurs du système comme la pompe et les LEDs. Pour cette partie de notre projet, nous avons choisi d'utiliser Next.js comme technologie de développement web. Nous avons appelé notre projet web « iGardenConnectWeb ».

Next.js est un framework open-source basé sur React qui permet de développer des applications web évolutives avec une grande efficacité. React est une bibliothèque Javascript* qui permet la conception d'interfaces utilisateurs.

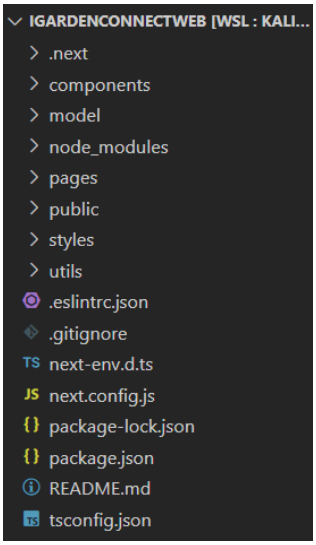



*Figure 23 - Structure d'un projet Next.js*

Sur la figure 23 ci-dessus, nous pouvons voir les différents dossiers qui constituent le projet. Le dossier « model » contient l'ensemble de nos classes qui sont des objets « View-Model » comme vu précédemment. Le dossier « pages » contient toutes les vues de l'applications. Le dossier « public » contient toutes les ressources qui seront disponibles en ligne comme les images du site. Le dossier « style » contient tous les fichiers CSS. Le dossier « utils » contient des fonctions utiles qui servent par exemple à la récupération de cookies. Le dossier components contient tous les composants du projet.

React se base sur un concept fondamental : tout ce qui est affiché à l'écran est considéré comme un composant. Un composant se défini comme une entité autonome et réutilisable qui peut s'imbriquer dans d'autres composants. Il peut aussi contenir des propriétés que nous verrons par la suite.

Cette orientation composant nous a permis de diviser notre application en petites parties facilement gérables.

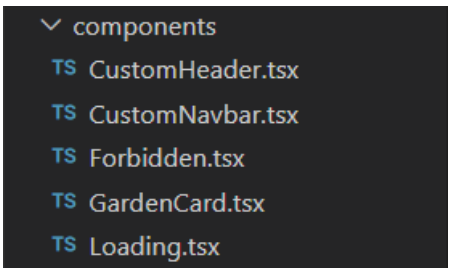



*Figure 24 - Composants du projet*

Pour illustrer ce qu'est un composant, nous pouvons prendre l'exemple de notre barre de navigation. À la place d'écrire le code de notre barre de navigation dans toutes les pages du projet. Nous en faisons un composant que nous avons appelé « CustomNavbar » comme nous pouvons le voir sur la figure 24 ci-dessus. Nous les utilisons dans chaque page comme si c'était une balise HTML. Les autres composants sont la page de chargement, la page de restriction d'accès, et un composant qui représente un jardin sur la page des jardins.

L'une des principales raisons pour lesquelles nous avons choisi Next.js est sa prise en charge de la génération de pages statiques et du pré-rendu, qui permettent de réduire considérablement le temps de chargement de nos pages. Cette fonctionnalité est particulièrement importante pour un projet de jardin connecté, car il peut y avoir un grand nombre de données à afficher en temps réel. Nous avons choisi une mise à jour des données toutes les dix secondes.

Une autre raison est que nous n'avions jamais utilisé de framework Javascript pour faire une application Web. Par le passé, nous avons eu à créer des applications Web en utilisant des technologies comme le PHP, l'HTML, le CSS et de simples fichiers Javascript. C'était donc un vrai défi pour nous car nous avons dû prendre en main cette technologie rapidement et la mettre en application.

Pour nous faciliter la tâche quant au design de l'application, nous avons utilisé Boostrap qui est une bibliothèque de composants « prêts à l'emploi » comme des boutons, grilles et autres modèles de mise en page. Cela nous a permis de gagner du temps.

Le système de plugins Next.js permet de personnaliser et d'étendre facilement les fonctionnalités de l'application web. Par exemple, nous avons utilisé le plugin React-Bootstrap qui nous a permis de directement utiliser des composants Boostrap dans la syntaxe React qui est orientée composant.

Cela permet une plus grande flexibilité dans le développement de l'interface utilisateur, cela peut nous permettre par exemple de répondre plus rapidement aux besoins des utilisateurs et d'adapter facilement l'application web en fonction des technologies dans le cas où continuerions à développer cette application.

De plus, l'écosystème de développement de Next.js est très actif ; avec une large communauté de développeurs et de nombreuses ressources en ligne pour aider à résoudre les problèmes rencontrés lors du développement. Cela permet une plus grande rapidité de développement et une meilleure qualité de code.

## 2. Présentation générale de l'interface utilisateur

L'interface utilisateur de « iGardenConnect » est organisée en plusieurs pages, chacune permettant à l'utilisateur de réaliser des actions spécifiques. Nous allons vous présenter les principales pages avant d'expliquer les fonctionnalités de Next.js qui nous ont permis de les réaliser.

Quand l'utilisateur écrit l'URL « http://localhost:3000 », il accède à la page d'accueil du site Web visible sur la figure 25 ci-dessous. Bien sûr, dans une situation idéale où le site serait déployé en ligne, il y accèderait avec l'URL « https://iGardenConnect.fr » par exemple.


*Figure 25 - Page d'accueil*

L'utilisateur a ensuite la possibilité de se connecter ou de créer son compte s'il n'en a pas en cliquant en haut à droite de la barre de navigation comme nous pouvons le voir ci-dessus.

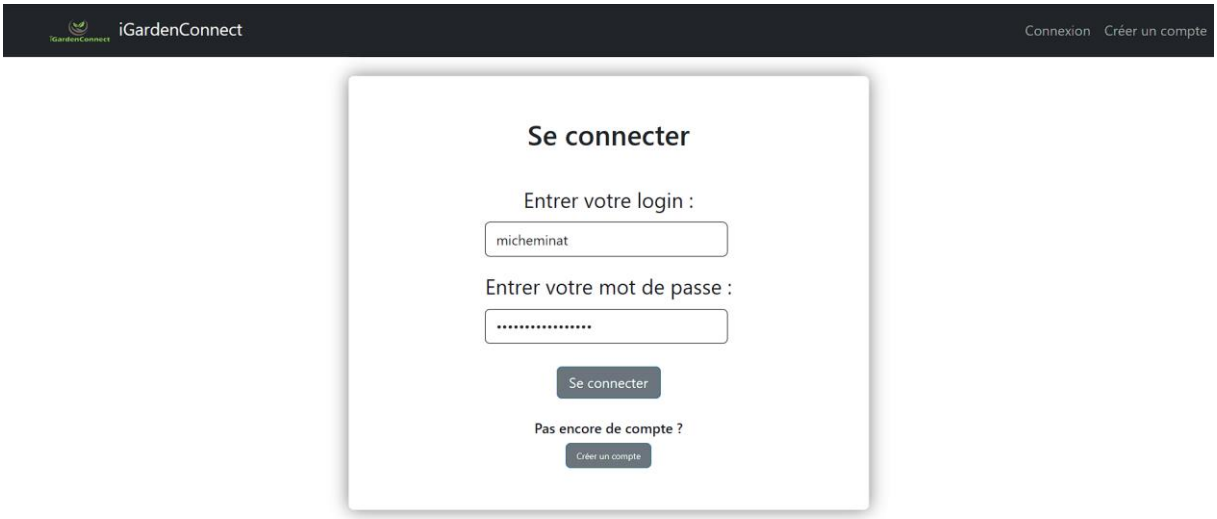



*Figure 26 - Page de connexion*

Les pages de connexion et de création de compte sont assez similaires donc nous ne montrerons que la page de connexion (figure 26). Nous détaillerons la création de compte et la connexion par la suite dans la partie « Sécurité ». Dès lors que l'utilisateur se connecte il accède à de nouvelles fonctionnalités. En effet, il peut désormais consulter la liste de ses jardins comme nous pouvons le voir dans la barre de navigation sur la figure ci-dessous.

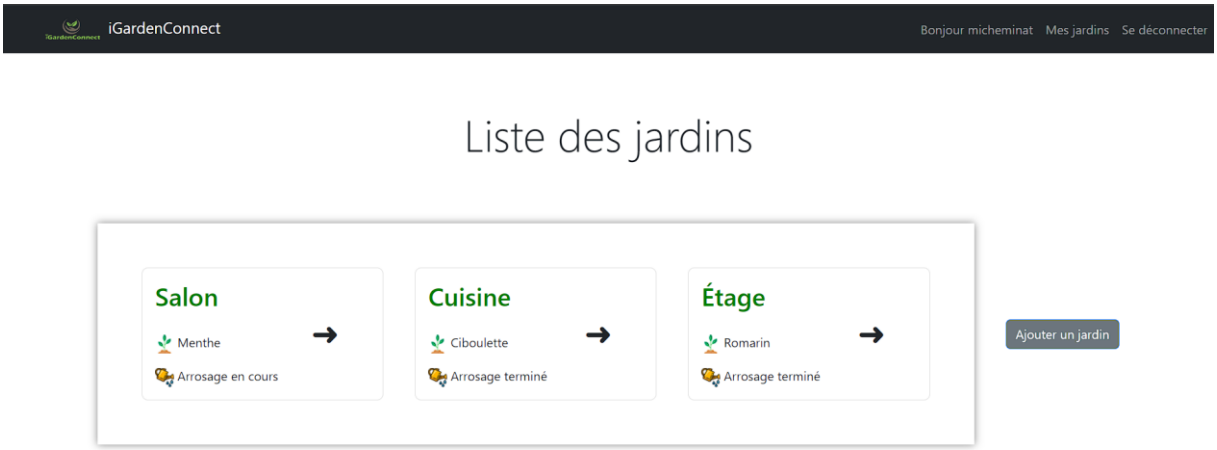



*Figure 27 - Page d'affichage des jardins*

Sur la figure 27 ci-dessus, nous pouvons voir que l'utilisateur « micheminat » comptabilise trois jardins différents dans sa maison. Nous pouvons observer qu'il élève de la menthe, de la ciboulette ainsi que du romarin. De plus, nous pouvons voir que l'arrosage est en cours dans le jardin du salon alors qu'il est terminé pour les jardins de la cuisine et de l'étage.

Si l'utilisateur vient d'acheter un nouveau jardin, il peut décider de l'ajouter à son compte pour avoir accès aux données des capteurs.

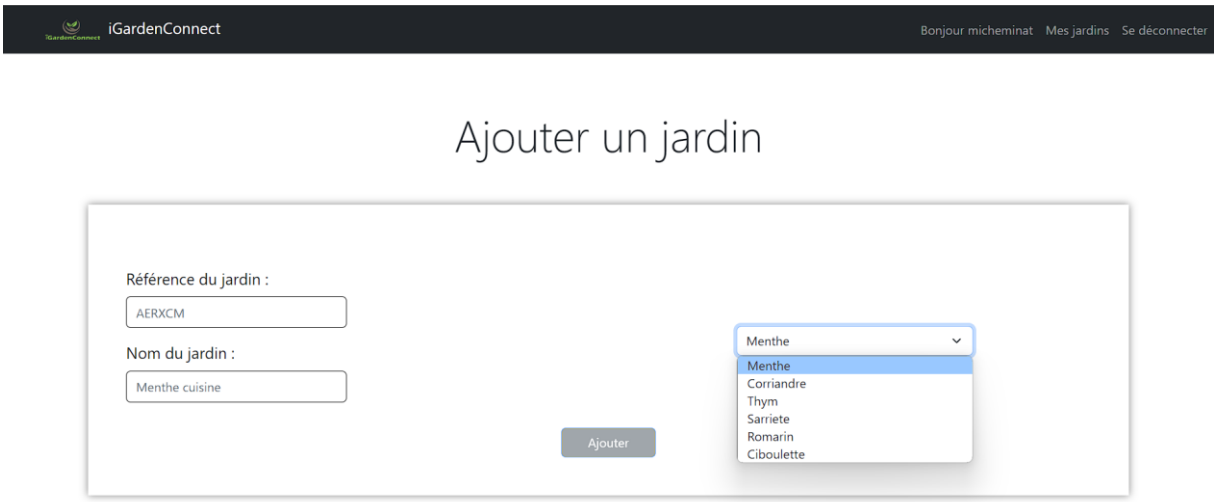



*Figure 28 - Page d'ajout d'un jardin*

Nous pouvons voir sur la figure 28 ci-dessus que l'utilisateur doit rentrer la référence du jardin. Nous partons du principe que la référence est unique et qu'elle est indiquée sur le jardin lors de son achat. Il peut aussi lui donner un nom et choisir une plante parmi une liste prédéfinie. Dès que le jardin est ajouté, il sera directement visible dans la liste des jardins.

Ensuite, si l'utilisateur clique sur un jardin il accède au détail de tous les capteurs.

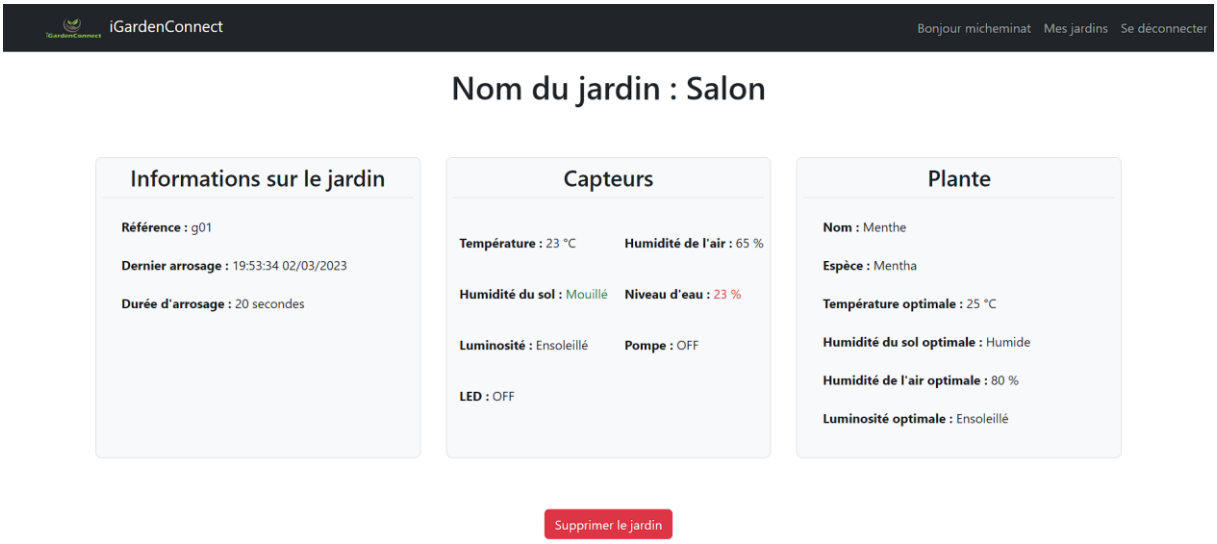



*Figure 29 - Page d'ajout d'un jardin*

Si nous revenons sur la figure 27, nous pouvons voir que le jardin du salon était en train d'être arrosé. Ici sur la figure 29, après avoir cliqué sur le jardin, nous pouvons voir que la pompe est éteinte. Nous avons directement accès sur la gauche à la date du dernier arrosage ainsi que la durée de cet arrosage. Dans la partie capteurs nous pouvons voir différentes informations importantes. Nous voyons que l'état du sol est mouillé et que le niveau d'eau du réservoir est assez bas. Ce qui est assez logique car l'arrosage vient juste d'être effectué. Le niveau d'eau en rouge informe l'utilisateur qu'il va bientôt devoir remplir le réservoir. De plus, sur la droite nous avons différentes informations importantes sur les conditions de vie idéale de la plante élevée. Enfin, si l'utilisateur décide de se séparer de son jardin connecté, il peut décider de le supprimer de son interface Web.

## 3. Utilisation des hooks et des méthodes asynchrones

L'utilisation de Next.js pour le développement de notre application « iGardenConnectWeb » nous a permis de bénéficier de nombreuses fonctionnalités avancées fournies par React ; les hooks et les méthodes asynchrones*. Ces fonctionnalités sont essentielles pour gérer efficacement l'état des composants et pour récupérer des données à partir de sources externes comme notre API « iGardenConnectAPI ».

### *a) Les hooks*

Les hooks sont des fonctions spéciales de React qui permettent de gérer l'état et le cycle de vie des composants. Dans le projet, nous avons utilisé les deux hooks les plus courants tels que useState et useEffect.

Le hook useState permet de définir et de modifier l'état d'un composant en le stockant dans une variable d'état.

```
const [plant, setPlant] = useState(new PlantVM());
```

*Figure 30 - Exemple d'un hook useState*

Par exemple comme nous pouvons le voir sur la figure 30, si nous souhaitons stocker une plante récupérée depuis une API, nous pouvons utiliser useState pour définir la variable d'état « plant ». La fonction « setPlant » permet de modifier cette variable d'état.

Le hook useEffect permet de déclencher une action lorsque l'état d'un composant change. Par exemple, si nous voulons récupérer un utilisateur depuis notre API et ensuite utiliser cette donnée, nous pouvons utiliser useEffect pour lancer la requête API lorsque le composant est monté. Le useEffect sera exécuté à chaque fois que la valeur de la variable passée en deuxième paramètre change. Si cette variable ne change jamais alors il ne sera exécuté qu'une seule fois au montage du composant.

```
useEffect(() => {
  getUserByToken().then(user => {
    setUser(user);
    setLoading(false);
  })
    .catch(() => {
      setLoading(false);
    })
}, [])
```

*Figure 31 - Exemple d'un hook useEffect*

Dans cet exemple sur la figure 31, nous voulons récupérer un utilisateur en fonction d'un jeton de connexion au chargement d'une page pour vérifier s'il est autorisé à accéder à la ressource. Nous souhaitons le faire qu'une seule fois alors nous passons une liste vide « [] » car elle ne changera jamais en tant que seconde valeur de dépendance pour que le hook ne soit réalisé qu'au chargement initial du composant.

L'utilisation de ces hooks permet une gestion efficace de l'état des composants et de leur cycle de vie, ainsi qu'une mise à jour dynamique de l'interface utilisateur en fonction des actions effectuées.

### *b) Les méthodes asynchrones*

L'utilisation des méthodes asynchrones a été indispensable à notre projet car elles permettent de gérer les appels à des ressources externes ; dans notre cas « iGardenConnectAPI ». JavaScript étant asynchrone, nous avons pu effectuer des appels à l'API sans bloquer l'interface utilisateur ce qui améliore considérablement son expérience car cela lui évite des ralentissements ou même des blocages.

Nous avons par exemple utilisé la méthode asynchrone fetch pour faire des requêtes à notre API.

```
fetch(`http://localhost:5241/api/Plant`)
  .then((response) => {
    if (!response.ok) {
      return response.text().then((errorMessage) => {
        throw new Error(errorMessage);
      });
    }
    return response.json();
  })
  .then((plantsdata) => {
    setPlants(plantsdata);
  })
  .catch((error) => {
    setError(error.message);
  });
```

***Figure 32 - Exemple d'une requête fetch***

Sur la figure 32 ci-dessus, nous appelons la route de notre API qui nous permet de récupérer toutes les plantes. Cette requête fetch retourne une promesse ; une promesse est un objet qui représente une valeur qui peut être disponible immédiatement, dans le futur ou qui peut être indisponible.

« then() » est une méthode de promesse qui permet de spécifier une fonction à exécuter lorsque la promesse est résolue. Dans notre cas, si la requête s'est bien passée, nous mettons à jour notre variable qui va contenir la liste de plantes. « catch() » permet un traitement en cas d'erreur de la promesse.

Les méthodes asynchrones nous ont aussi permis de mettre à jour l'interface utilisateur en temps réel lorsque les données sont récupérées. Nous l'utilisons pour mettre à jour les données des capteurs sur la page de détail d'un jardin à un intervalle de temps régulier.

Cet intervalle est le même que celui que le jardin aura pour envoyer les données dans la base de données.

#### 4. TypeScript

TypeScript est un langage de programmation open-source qui est une surcouche de JavaScript. Il a été développé par Microsoft et il est de plus en plus utilisé dans les applications Web. Il a introduit les types statiques ce qui permet d'éviter des erreurs de typage qui sont faciles à détecter.

Dans notre projet de jardin connecté, nous avons utilisé TypeScript pour garantir la sécurité et la robustesse de notre code. Grâce au typage, nous avons pu éviter les bugs à l'exécution. De plus, TypeScript nous a permis d'utiliser les fonctionnalités modernes de Javascript comme les classes tout en étant sûr que notre code serait compatible avec les navigateurs.

### C. Sécurité :

#### 1. Bcrypt

La sécurité est un aspect essentiel de notre projet. Pour protéger les informations des utilisateurs, nous avons tout d'abord un format de mot de passe et nous avons utilisé des techniques de chiffrement pour stocker les mots de passe.

Plus précisément, nous avons utilisé la bibliothèque Bcrypt pour « hash » les mots de passe des utilisateurs avant de les stocker dans notre base de données.

Bcrypt est très sécurisée. C'est une fonction de hachage de mot de passe qui utilise une combinaison de salage* et de chiffrement pour générer une chaine de caractère unique à partir du mot de passe de l'utilisateur. Cet algorithme est très résistant aux attaques de dictionnaire* et aux attaques par force brute*. Il permet aussi d'ajuster la complexité du chiffrement en fonction des besoins des développeurs. Nous avons choisi 10 tours ce qui fait un bon entre-deux entre les contraintes de performances et de sécurité, c'est ce qui se fait dans la plupart des cas.

Cette technique empêche les attaquants d'accéder aux mots de passe en clair même s'ils réussissent à obtenir un accès non autorisé à la base de données.

## 2. JWT

En plus du chiffrement des mots de passe, nous avons mis en place un système de contrôle d'accès pour empêcher les utilisateurs non autorisés d'accéder à des pages ou des fonctionnalités qui ne leurs sont pas destinées. Pour cela, nous avons utilisé des jetons de session JWT.

JWT pour JSON Web Tokens est très utilisé pour gérer l'authentification des utilisateurs côté API. Lorsqu'un utilisateur se connecte, le serveur génère un jeton JWT qui va être renvoyé au client qui va le stocker localement dans un cookie. Ensuite, par exemple si l'utilisateur cherche à accéder à ses jardins, alors le client envoie le jeton JWT dans l'en tête d'authentification de la requête au serveur. L'API va alors vérifier la validité du jeton en vérifiant que la date d'expiration n'est pas dépassée et que la signature est la bonne. Nous utilisons une clé privée contenue dans l'API pour chiffrer les données. Si le jeton n'est pas valide, l'utilisateur est redirigé vers une page d'erreur appropriée.

Le fait d'utiliser JWT et Bcrypt nous a permis d'ajouter une réelle couche de sécurité à l'application. Ces technologies nous ont permis de gérer efficacement les sessions en nous évitant de sauvegarder l'état de la session sur le serveur.

## 3. HTTPS

Nous étions dans un environnement de développement donc nous avons travaillé en HTTP. Cependant il est important de prendre en compte l'utilisation de HTTPS pour la sécurité de notre application. HTTPS nous permet de garantir à l'utilisateur que les données envoyées entre le client et le serveur sont chiffrées. Nous sommes donc conscients de l'obligation de passer en HTTPS si jamais notre application web ainsi que notre API venaient à être déployés sur un serveur en ligne

## 4. CORS

Cross-Origin Resource Sharing (CORS) est un mécanisme de sécurité utilisé par les navigateurs web pour contrôler l'accès à des ressources partagées entre différents sites web. Il permet de restreindre les requêtes HTTP entre deux domaines différents afin de garantir que seuls les domaines autorisés peuvent accéder aux ressources de l'API. Les navigateurs web imposent des politiques de sécurité strictes pour éviter les attaques de type cross-site scripting (XSS) et cross-site request forgery (CSRF).

```
// Allow CORS
app.UseCors(builder => builder
    .WithOrigins("http://localhost:3000") // Add your allowed origins here
    .AllowAnyMethod()
    .AllowAnyHeader());
```

*Figure 33 - Règle CORS pour iGardenConnectAPI*

Dans le cas de notre projet, nous avons utilisé les méthodes CORS dans l'API pour autoriser l'accès uniquement à notre site web iGardenConnectWeb hébergé sur le même serveur que l'API. Nous avons configuré les en-têtes de réponse CORS pour permettre l'accès aux ressources de l'API uniquement à partir de notre domaine et bloquer les requêtes provenant de tout autre domaine.

Cela permet de réduire les risques de vulnérabilités de sécurité en limitant l'accès aux ressources sensibles de l'API uniquement aux domaines de confiance.

# VI. Bilan technique

Au terme de ces cinq mois de projet, les fonctionnalités que nous avons eu à créer fonctionnent. Nous avons testé l'application web et l'API sur différents appareils. Nous n'avons pas pu mettre en place des tests approfondis comme des tests unitaires par manque de temps.

Voici la liste des fonctionnalités qui ont été réalisées :

- La création, modification et suppression de jardins
- La création, modification et suppression d'une plante
- La création, modification, suppression de capteurs
- La possibilité d'un jardin de publier les valeurs des capteurs en base de données
- Une création de compte et une connexion sécurisée avec gestion de session
- Affichage en temps réel des données d'un jardin

Nous avons parfois rencontré des difficultés durant le projet. En effet, les technologies .NET Core et Next.js étaient nouvelles pour nous. Nous avons donc eu besoin d'un temps d'adaptation et de compréhension avant de pouvoir être effectifs. Aussi, le design n'étant pas notre point fort, nous nous sommes concentrés sur le côté fonctionnel de l'application Web. De plus, le fait de devoir tout créer de la conception au développement en parallèle de nos cours, partiels et recherches de stage nous a appris à nous organiser et à gérer notre temps de manière efficace.

Enfin, nous avons quelques pistes d'amélioration en tête :

- Un choix plus grand de plantes, une meilleure personnalisation
- La possibilité d'activer/désactiver la pompe et les LEDs depuis le site Web
- Une partie administrateur qui nous permettrait de gérer notre application
- Une gestion de plus de failles de sécurité Web, le passage en HTTPS de l'API et du site Web
- Ajouter des fonctionnalités pour rendre le site plus interactif, des animations, un meilleur design
- Créer une application mobile qui utiliserait l'API de la même manière que le site Web

# VII. Conclusion

iGardenConnectAPI et iGardenConnectWeb nous ont permis de mettre en pratique des connaissances théoriques acquises au cours de notre formation. La méthode agile* utilisée pour la conception et le développement des fonctionnalités a été très bénéfique, permettant une meilleure communication et collaboration entre les membres de l'équipe. De nombreux outils ont été utilisés, tels que Git pour la gestion de versions, GanttProject pour la planification et l'organisation des tâches, ou encore Swagger pour la documentation de l'API.

La prise en compte de notions de cybersécurité, comme l'utilisation de Bcrypt pour chiffrer les mots de passe et la gestion de sessions avec JWT, a été importante pour garantir la sécurité des données des utilisateurs. Enfin, le développement logiciel en lui-même a été très enrichissant, nous permettant d'apprendre de nouvelles technologies et de renforcer nos compétences de programmation.

Même si nous n'avons pas participé directement à la création du jardin connecté et à la mise en place de tous les capteurs, nous avons suivi de très près son évolution et sommes très fiers du résultat qu'ils ont obtenu. Des photos du jardin terminé sont disponibles en annexe sur les figures 39, 40, 41.

Nous avons atteint la plupart nos objectifs fixés en début d'année. Cependant, la liste des possibles améliorations est longue. En effet, la possibilité pour un utilisateur d'activer/désactiver la pompe ou les LEDs, une meilleure sécurité des applications et une partie administrateur seraient les premières fonctionnalités que nous ajouterions au projet.

En conclusion, iGardenConnectAPI et iGardenConnectWeb ont été des projets très enrichissants. Ils nous ont permis de mettre en pratique des connaissances acquises en cours tout en développant de nouvelles compétences. Nous sommes satisfaits du résultat final et espérons que ce projet sera utile à des utilisateurs souhaitant gérer leurs jardins de manière simple et efficace.

# VIII. Abstract

From September 5th until March 3rd, we had to create a project as part of our second-year tutored project called “Easy data”. This project aimed to create a system that manages and monitors a garden using sensors and automated processes.

Maintaining a garden can be a time-consuming and challenging task, especially for those who are not familiar with plants. It can be difficult to know what type of plants to grow, how often to water them, and how much sunlight they need. In addition, it can be challenging to monitor the growth of plants and identify potential issues such as overwatering, underwatering, or diseases.

First, we started by conducting extensive research into the needs of gardeners and the types of technology that could be used to create an efficient and user-friendly web-application.

We created the SQL Server database we were going to need. We started to work on the project, by creating the back end of the application in an API named iGardenConnectAPI. We created all the logic for every feature before working on the user interface. The back end which was coded in C# can be resumed by the acronym “CRUD” which means Create, Read, Update and Delete. We used a framework called Entity Framework Core to facilitate the queries to the database. The front end was created using Next.js framework, we used Bootstrap to ease the design development.

While the application is already functional, there are several new features that the team may consider adding in the future. One potential addition could be the ability for the application to suggest specific plants or types of plants based on the user's location, the season, and their past planting history. One aspect of the design that could be improved is the visual presentation. In fact, a better web-design could make the application more intuitive and user-friendly, while also improving its overall aesthetic appeal. Also, the design could be enhanced with more visuals, such as photos and graphics of the plants, to provide a more immersive and engaging experience.

One of the most challenging aspects of the project was the fact that the team had to go through the entire product development process, from creation to testing, all within a limited time frame.

This project was helpful to develop our skills on design and application architecture. It has also reinforced our abilities to work in a team because this process required a significant amount of collaboration and communication between team members.

# IX. Lexique

**Microservices :** Une architecture de développement de logiciel dans laquelle une application est divisée en services distincts et indépendants les uns des autres, permettant une évolutivité et une flexibilité accrues.

**API :** Interface de programmation d'application - un ensemble de protocoles et de règles utilisés pour construire des applications logicielles en définissant comment les différents composants doivent communiquer entre eux.

**Front-end :** Partie visible et interactive d'une application logicielle qui est présentée à l'utilisateur final.

**Framework :** Un ensemble de bibliothèques de code et d'outils qui permettent de développer plus facilement et rapidement des applications logicielles en fournissant des fonctionnalités de base.

**Open-source :** Un logiciel dont le code source est accessible à tous et qui peut être utilisé, modifié et distribué gratuitement.

**Mapping :** La correspondance entre les données stockées dans une base de données et les objets utilisés dans un programme.

**Débogage :** Le processus de recherche et de correction des erreurs dans le code d'un programme.
**SQL :** Structured Query Language - un langage de programmation utilisé pour interagir avec les bases de données relationnelles.

**React :** Une bibliothèque JavaScript open-source utilisée pour construire des interfaces utilisateur interactives et réactives.

**HTTP :** Hypertext Transfer Protocol - le protocole utilisé pour transférer des données sur le World Wide Web.

**Gitlab :** Une plateforme web open-source pour la gestion du cycle de vie des applications logicielles.

**Branche :** Une copie du code source d'un projet qui peut être modifiée et testée sans affecter le code principal.

**UML :** Unified Modeling Language - un langage de modélisation graphique utilisé pour spécifier, visualiser et documenter les systèmes logiciels.

**Jalon :** Un point de référence dans le temps pour mesurer l'avancement d'un projet.

**Critique :** Une étape dans le processus de développement de logiciels qui nécessite une attention particulière en raison de son importance pour le projet.

**Clés primaires :** Un ou plusieurs champs dans une table de base de données qui identifient de manière unique chaque enregistrement.

**Clés étrangères :** Une colonne dans une table de base de données qui identifie de manière unique une ligne dans une autre table de base de données.

**Swagger :** Un outil open-source pour la documentation des API RESTful.

**Méthodes :** Les fonctions ou procédures qui sont utilisées pour effectuer des opérations dans un programme.

**Ef Core Power Tools :** Un outil de développement pour Entity Framework Core qui fournit des fonctionnalités pour faciliter le développement de logiciels.

**URL :** Uniform Resource Locator - l'adresse qui identifie de manière unique une ressource sur le World Wide Web.
Asynchrones : Des opérations ou des fonctions qui s'exécutent indépendamment les unes des autres, permettant à d'autres processus de se poursuivre en même temps.

**Javascript :** Langage de programmation utilisé dans le développement Web coté client.

**Méthode agile :** consiste à diviser le projet en courte période (que nous appellerons sprint). Ensuite le client est très impliqué afin d'obtenir un résultat qui correspond à ses volontés.

# X. Bibliographie

[1] 2013, Mark Massé, Rest API, Design RuleBook.
Cet ouvrage nous donne les clés pour développer des APIs qui respectent les règles REST. Il nous permet de comprendre comment utiliser les verbes et les routes pour les applications Web qui utilisent des API.

# XI. Webographie

[Moqups] https://app.moqups.com/ (Consulté le 07/11/2022)

[mobiskill] https://mobiskill.fr/blog/conseils-emploi-tech/net-core-vs-net-framework-quelles-differences/ (Consulté le 25/11/2022)

[Microsoft] https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15 (Consulté le 12/12/2022)

[Microsoft] https://www.microsoft.com/en-us/sql-server/sql-server-downloads (Consulté le 13/12/2022)

[code-maze] https://code-maze.com/net-core-web-development-part5/ (Consulté le 19/12/2022)

[developpez] https://pmusso.developpez.com/tutoriels/dotnet/entity-framework/introduction/ (Consulté le 05/01/2023)

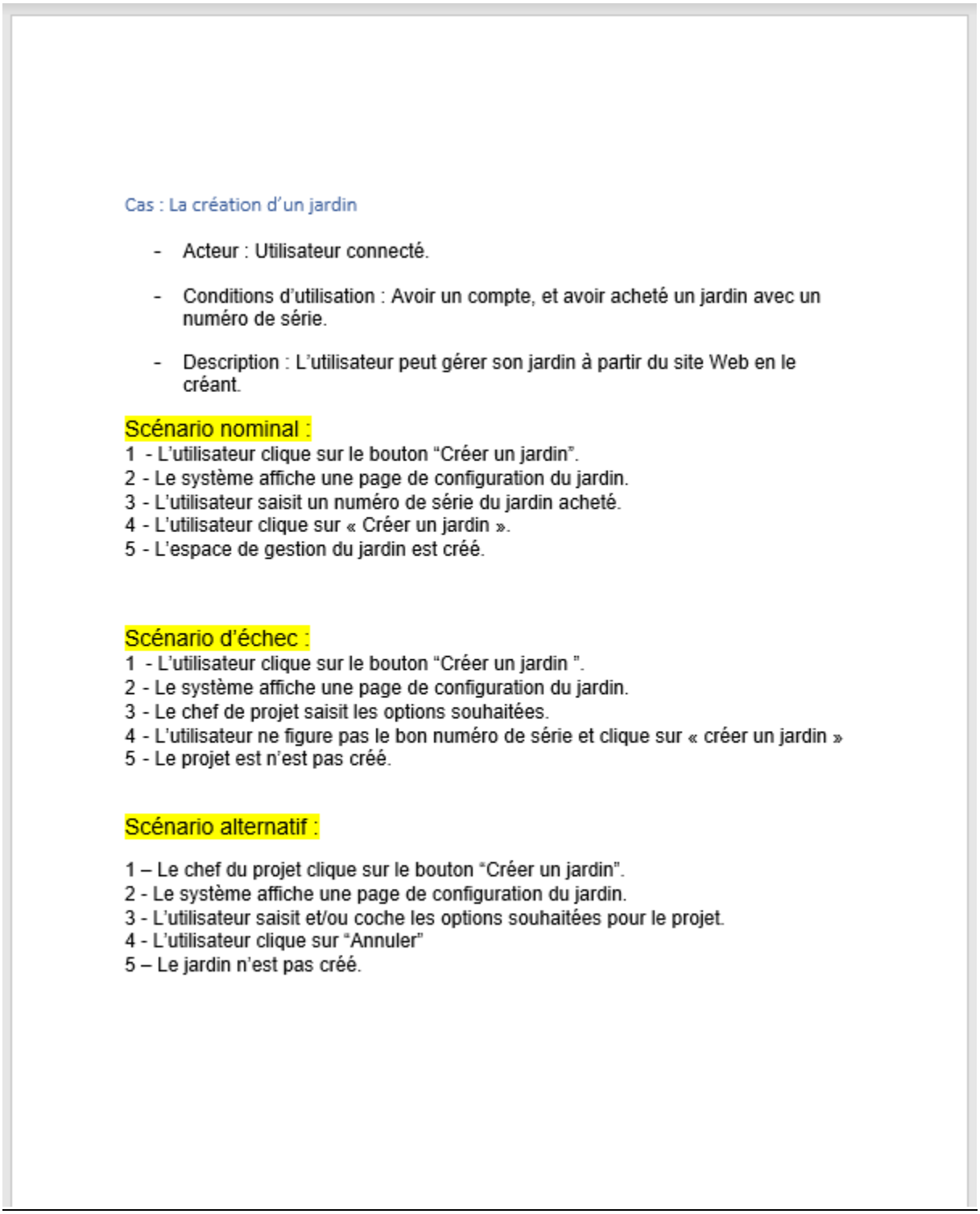
[Neoito] https://www.neoito.com/blog/create-a-website-with-next-js/ (Consulté le 02/02/2023)

[medium] https://medium.com/@david.nissimoff/next-js-meets-asp-net-core-a-story-of-performance-and-love-at-long-tail-41cf9231b2de (Consulté le 05/02/2023

[nextjs] https://nextjs.org/docs/api-reference/create-next-app (Consulté le 01/02/2023

[mdbootsrap] https://mdbootstrap.com/docs/react/utilities/spacing/ (Consulté le 15/02/2023)

# XII. Annexes



Cas : La création d'un jardin

- Acteur : Utilisateur connecté.
- Conditions d'utilisation : Avoir un compte, et avoir acheté un jardin avec un numéro de série.
- Description : L'utilisateur peut gérer son jardin à partir du site Web en le créant.

Scénario nominal :

1 - L'utilisateur clique sur le bouton "Créer un jardin".
2 - Le système affiche une page de configuration du jardin.
3 - L'utilisateur saisit un numéro de série du jardin acheté.
4 - L'utilisateur clique sur « Créer un jardin ».
5 - L'espace de gestion du jardin est créé.

Scénario d'échec :

1 - L'utilisateur clique sur le bouton "Créer un jardin ".
2 - Le système affiche une page de configuration du jardin.
3 - Le chef de projet saisit les options souhaitées.
4 - L'utilisateur ne figure pas le bon numéro de série et clique sur « créer un jardin »
5 - Le projet est n'est pas créé.

Scénario alternatif :

1 – Le chef du projet clique sur le bouton "Créer un jardin".
2 - Le système affiche une page de configuration du jardin.
3 - L'utilisateur saisit et/ou coche les options souhaitées pour le projet.
4 - L'utilisateur clique sur "Annuler"
5 – Le jardin n'est pas créé.

*Figure 34 - Description du cas « Création d'un jardin »*

Cas : Consulter ses jardins

- Acteur : Tous les utilisateurs.
- Conditions d'utilisation : être identifié.
- Description : Les utilisateurs peuvent consulter les jardins

Scénario nominal :

1- L'utilisateur sélectionne un jardin dans la liste des jardins crées par l'utilisateur.
2– Le système répond en affichant la page du jardin que l'utilisateur a sélectionné.
3 – La page affiche toutes les données du jardin : température, humidité, lumière, niveau d'eau.

*Figure 35 - Description du cas « Consulter ses jardins »*

Cas : S'identifier

- Acteur : Utilisateurs non connectés.
- Conditions d'utilisation : avoir un compte existant, ou le créer.
- Description : L'identification permet de se connecter et de bénéficier des différentes fonctionnalités dédiées aux utilisateurs.

Scénario nominal :

1 - L'utilisateur clique sur le bouton "S'identifier".
2 - Le système affiche une fenêtre d'authentification.
3 - L'utilisateur saisit son identifiant, et son mot de passe.
4 - L'utilisateur confirme ses données.
5 - Le système vérifie l'authentification.
6 - L'utilisateur est connecté.

Scénario d'échec :

1 - L'utilisateur clique sur le bouton "S'identifier".
2 - Le système affiche une fenêtre d'authentification.
3 - L'utilisateur saisit un identifiant ou un mot de passe incorrect.
4 - L'utilisateur confirme ses données.
5 - Le système vérifie l'authentification.
6 - L'utilisateur n'existe pas ou le mot de passe est incorrect.
7 - Affichage "identifiant ou mot de passe incorrect".

Scénario alternatif :

1 - L'utilisateur clique sur le bouton "S'identifier".
2 - Le système affiche une fenêtre d'authentification.
3 - L'utilisateur saisit son identifiant ou son mot de passe.
4 - L'utilisateur clique sur "Annuler"
5 - L'utilisateur n'est pas connecté.

*Figure 36 - Description du cas « S'identifier »*

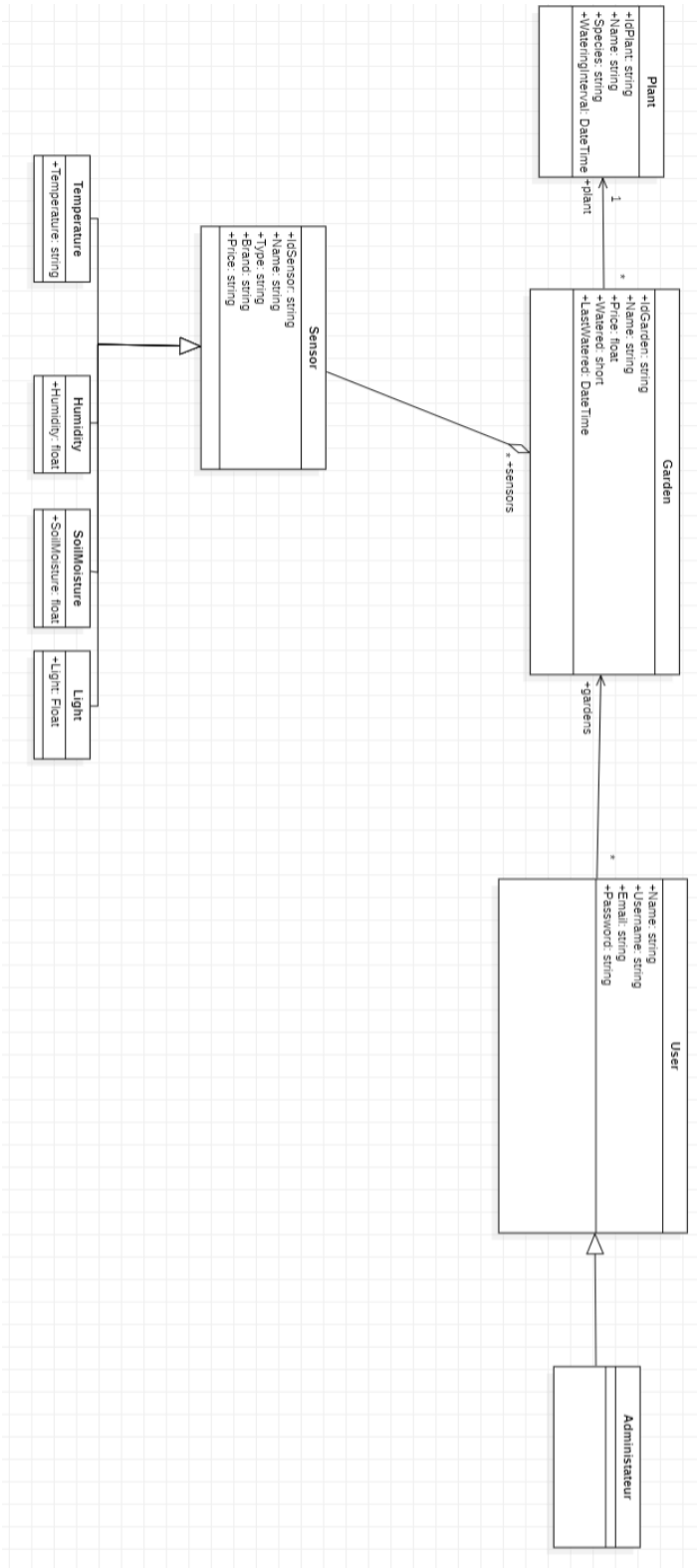



*Figure 37 - Diagramme de classe*

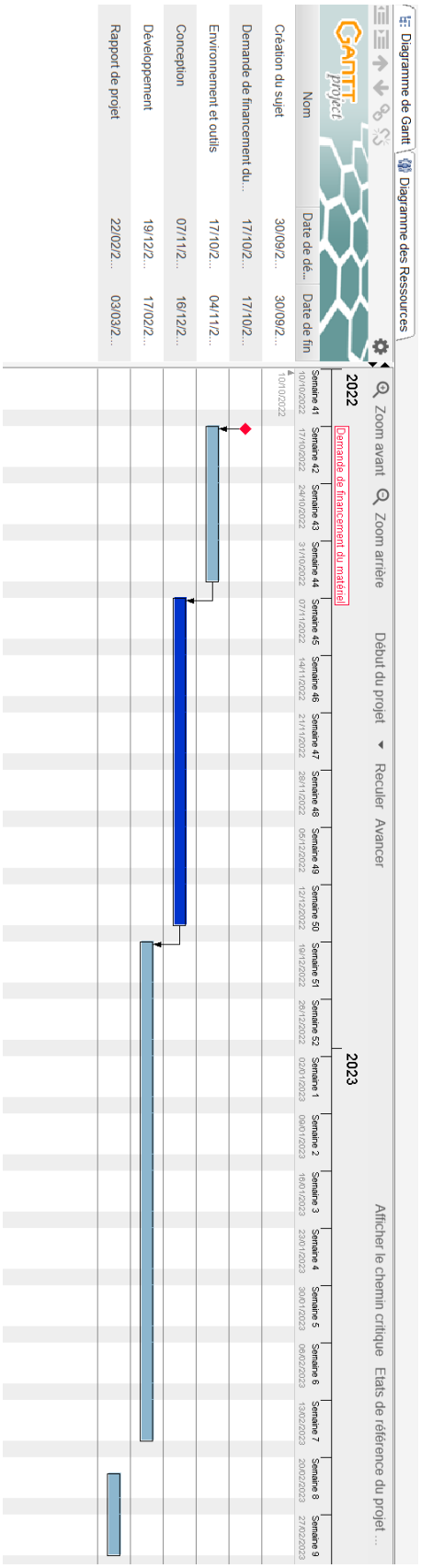



*Figure 38 - Diagramme de Gantt prévisionnel*


*Figure 39 - Jardin connecté terminé*


*Figure 40 - Jardin connecté avec LEDs allumées*


*Figure 41 - Jardin connecté avec pompe activée*