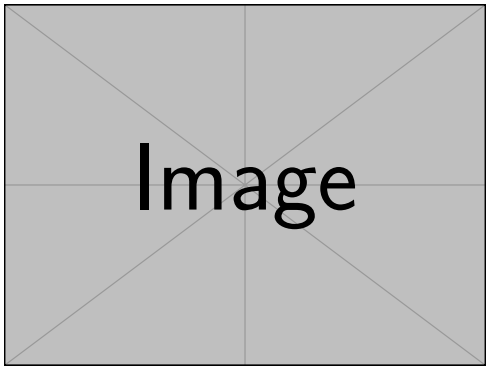
MediPlanner

Documentation Technique



Version : V0.1

Rédigé par : Axel LELONG
Projet EPITA 2024
Institut Curie

17 janvier 2025

Table des matières

1 Introduction à la Partie C#

2 Les classes

2.1 Introduction

Le code présenté définit une classe `Center`, qui représente un centre médical, incluant les machines, les ressources et les localisations. La classe est construite avec diverses propriétés et méthodes permettant de gérer ces éléments.

2.2 Classe `Center`

La classe `Center` représente un centre médical avec plusieurs machines, ressources et localisations. Elle contient plusieurs propriétés et méthodes pour gérer les machines, récupérer des machines spécifiques, et gérer les préférences de localisation.

2.2.1 Propriétés

- `Id` : Identifiant unique du centre.
- `Name` : Nom du centre.
- `Machines` : Liste des machines associées au centre.
- `Ressources` : Liste des ressources humaines associées au centre.
- `Localisations` : Liste des localisations médicales disponibles dans le centre.

2.2.2 Constructeurs

- Le premier constructeur permet de créer un centre avec un nom spécifique, initialisant les autres propriétés comme vides ou par défaut.
- Le second constructeur est une valeur par défaut pour un centre nommé `StCloud`, avec des machines et des localisations préconfigurées.

2.2.3 Méthodes

- `AddMachine(Machine machine)` : Ajoute une machine au centre.
- `GetMachineByName(string name)` : Récupère une machine par son nom.
- `GetLocaByName(string loca)` : Récupère le type de localisation par son nom.
- `GetMachinesByPreference(LocalizationType localization)` : Récupère les machines triées par préférence pour un type de localisation donné.
- `GetMostOptimalMachine(VarianContext context, LocalizationType localization, Tuple<int, int> Champs, DateTime date, int weeks)` : Récupère la machine optimale en fonction des préférences et de la disponibilité.
- `DisplayPreference(Preferences pref)` : Affiche la préférence d'une machine sous forme textuelle.
- `ColorPrefence(Preferences pref)` : Retourne la couleur associée à une préférence donnée.
- `DisplayTreatmentTable()` : Affiche un tableau des préférences des machines pour chaque localisation.

2.2.4 Exemple d'utilisation

Dans la classe `Center`, des machines comme `TRUEBEAM`, `NOVALISTX` et `HALCYON SC` sont ajoutées avec des préférences associées à chaque localisation, afin de déterminer quelles machines sont les plus appropriées pour un traitement donné. Les préférences sont gérées à l'aide de l'énumération `Preferences`.

2.3 Énumération `Preferences`

L'énumération `Preferences` est utilisée pour définir les préférences de traitement. Elle possède quatres valeurs possibles :

- `PRIORITE` : Indique que la localisation est prioritaire pour un traitement.
- `POSSIBLE` : Indique que la localisation est possible pour un traitement.
- `AEVITER` : Indique que la localisation doit être évitée.
- `NON` : Indique que la localisation ne pas être traitée.

2.4 Énumération `LocalizationType`

L'énumération `LocalizationType` est utilisée pour représenter différents types de localisations médicales disponibles dans le centre. Ces types incluent des catégories comme `SeinLocal`, `Hematologie`, `Urologie`, etc.

2.5 Classe `Event`

Le code présente la classe `Event`, qui gère les événements en enregistrant des informations telles que la date de début, de fin, le titre, l'ID, et d'autres propriétés supplémentaires. La classe `ExtendedProps` est utilisée pour stocker des informations supplémentaires sur chaque événement, telles que sa source et son statut.

La classe `Event` représente un événement avec plusieurs propriétés permettant de stocker des informations pertinentes pour le processus de gestion des événements.

2.5.1 Propriétés

- `Title` : Le titre de l'événement (chaîne de caractères).
- `Start` : La date et l'heure de début de l'événement (de type `DateTime`).
- `End` : La date et l'heure de fin de l'événement (de type `DateTime`).
- `AllDay` : Indicateur booléen qui indique si l'événement dure toute la journée.
- `Id` : L'ID unique de l'événement (chaîne de caractères).
- `ExtendedProps` : Un objet de type `ExtendedProps`, qui contient des propriétés supplémentaires liées à l'événement (source et statut).

2.5.2 Constructeurs

- Le constructeur par défaut (`Event()`) : Permet de créer un événement sans spécifier de valeurs initiales.
- Le constructeur avec paramètres (`Event(string title, DateTime start, DateTime end, bool allDay, string id, string source, string status)`) : Permet de créer un événement avec des valeurs spécifiques pour le titre, les dates, la durée (journée entière), l'ID, ainsi que la source et le statut via l'objet `ExtendedProps`.

2.6 Classe `ExtendedProps`

La classe `ExtendedProps` contient des informations supplémentaires qui ne sont pas directement liées aux propriétés principales d'un événement, mais qui sont importantes pour son traitement.

2.6.1 Propriétés

- `Source` : La source de l'événement (chaîne de caractères).
- `Status` : Le statut actuel de l'événement (chaîne de caractères).

2.6.2 Constructeurs

- Le constructeur par défaut (`ExtendedProps()`) : Permet de créer une instance sans initialisation de valeurs.
- Le constructeur avec paramètres (`ExtendedProps(string source, string status)`) : Permet d'initialiser les propriétés `Source` et `Status`.

2.7 Classe `Machine`

La classe `Machine` représente une machine utilisée dans le cadre du processus de traitement. Elle contient des informations relatives à la machine, telles que son nom, son type, ses spécifications locales, et les tailles maximales de traitement possibles. Cette classe permet de gérer les machines et les spécifications associées à chaque localisation où elles sont utilisées.

2.7.1 Propriétés

- `Id` : L'ID unique de la machine (chaîne de caractères).
- `Name` : Le nom de la machine (chaîne de caractères).
- `MachineType` : Le type de la machine (de type `MachineType`, qui est une énumération).
- `Localizations` : Un dictionnaire associant des types de localisation (`LocalizationType`) à des tuples contenant les préférences et un indicateur booléen pour savoir si la localisation est officielle.
- `Champs` : Un tuple contenant deux entiers représentant les tailles maximales de traitement de la machine.

2.7.2 Constructeurs

- Le constructeur par défaut (`Machine()`) : Permet de créer une instance de `Machine` sans spécifier de valeurs initiales.
- Le constructeur avec paramètres (`Machine(string name, MachineType MachineType, Tuple<int, int> champs)`) : Utilisé pour créer une machine avec un nom, un type de machine, et des tailles de champs prédéfinies. Ce constructeur est utilisé spécifiquement pour un site durci (St-Cloud).

2.7.3 Méthodes

- `AddLocalization(LocalizationType localization, Preferences pref, bool Officiel)` : Cette méthode permet d'ajouter une localisation traitable à la machine, avec des préférences associées et un indicateur pour savoir si la localisation est officielle.
- `GetMachineName()` : Cette méthode retourne le nom de la machine.
- `GetLocalizationPref(LocalizationType localization)` : Cette méthode retourne la préférence d'une localisation spécifique sur la machine. Si la localisation n'est pas trouvée, elle retourne `Preferences.NON`.
- `GetLocalizationOffcial(LocalizationType localization)` : Cette méthode retourne un booléen indiquant si une localisation est officielle sur la machine. Si la localisation n'est pas trouvée, elle retourne `false`.

2.7.4 Résumé

La classe `Machine` permet de définir et gérer les machines, en enregistrant leur nom, leur type, les localisations traitables et les tailles maximales des champs. Elle inclut des méthodes pour ajouter des localisations, obtenir des préférences spécifiques et savoir si une localisation est officielle. Cela permet de gérer de manière souple les machines et leur utilisation dans différents contextes.

3 Le projet

Le projet **MediPlanner** est divisé en trois sous-projets principaux, chacun ayant un rôle spécifique :

3.1 Structure des Sous-Projets

— **MediCore** : Contient les contrôleurs, responsables de la gestion des différentes fonctionnalités, telles que la gestion des sites, des plannings, et des statistiques.
— **MediData** : Inclut les modèles générés automatiquement, représentant la structure des données utilisées dans l'application.
— **MediPlanner** : Gère la logique interne de l'application, permettant de coordonner les interactions entre les contrôleurs et les modèles.

4 MediCore : Les Contrôleurs

Le sous-projet **MediCore** regroupe les contrôleurs suivants :

4.1 AdminController : Gestion des Sites

Le **AdminController** est responsable de la gestion des sites, permettant d'ajouter, modifier ou supprimer des informations sur les centres de traitement.

Le contrôleur `AdminController` implémente plusieurs fonctionnalités pour la gestion des centres, des ressources humaines et des machines dans l'application MediPlanner.

1. **Création de centres :**
   — `CreateCentre` : Crée un nouveau centre avec un nom spécifié.
2. **Récupération des centres :**
   — `GetCentres` : Retourne tous les centres enregistrés.
   — `GetCenter` : Retourne les informations d'un centre spécifique, selon son ID.
3. **Suppression de centres :**
   — `DeleteCentre` : Supprime un centre par son ID.
4. **Édition des ressources humaines d'un centre :**
   — `EditCentre` : Permet de modifier les ressources humaines d'un centre en utilisant une liste de ressources.
5. **Édition des machines d'un centre :**
   — `EditCentreMachines` : Permet d'ajouter des machines à un centre en utilisant une liste de machines.
6. **Édition de l'affinité des machines (préférences et priorités) :**
   — `EditMachinePref` : Permet de modifier l'affinité des machines, comme les préférences et priorités pour différents types de localisation.
7. **Récupération des machines :**
   — `GetMachines` : Retourne toutes les machines connues dans la base de données.
   — `GetMachineCenter` : Retourne une machine spécifique d'un centre.
8. **Récupération des ressources humaines :**
   — `GetRessources` : Retourne toutes les ressources humaines connues dans la base de données.

9. **Importation de centres via une configuration externe :**
   — `ImportCentres` : Permet d'importer une liste de centres via une configuration externe.

4.1.1 Points clés

— Le contrôleur `AdminController` utilise le service `CenterService` pour gérer les centres et le contexte `VarianContext` pour interagir avec la base de données.
— Il permet d'ajouter, modifier et supprimer des centres et des machines tout en maintenant des logs pour le suivi des actions effectuées.
— Le contrôle d'accès et la gestion des erreurs sont mis en place pour assurer un bon fonctionnement.

4.2 PlanningController : Gestion des Plannings

Le **PlanningController** permet de gérer les plannings des machines et des patients, en assurant une allocation optimale des ressources.

Les fonctions du `PlanningController` permettent de gérer les plannings des machines, des patients et des traitements dans le projet `MediCore`. Voici un résumé détaillé de ces fonctions.

- `HelloPage()` :
  - **But :** Envoie un message d'accueil sur la page d'initialisation pour des tests ou des configurations initiales.
  - **Fonctionnement :** Lorsqu'un utilisateur accède à l'URL de cette méthode (`/`), elle répond avec un message `"hello"` et enregistre un log d'information.
- `GetSchedule()` :
  - **But :** Récupère le planning complet pour une machine spécifique à partir d'une date de début donnée.
  - **Fonctionnement :** Prend comme paramètres le nom de la machine et la date de début. Interroge la base de données pour récupérer l'emploi du temps de la machine via la fonction `GetScheduleActivity` et renvoie les résultats sous forme d'une réponse HTTP.
- `GetPreferredMachine()` :
  - **But :** Récupère la machine préférée pour un traitement basé sur la localisation, la taille de la tumeur, et la date de début du traitement.
  - **Fonctionnement :** Recherche dans les machines du centre celle qui correspond le mieux aux besoins du patient, en fonction de la localisation et de la taille de la tumeur.
- `GetCompatibleMachines()` :
  - **But :** Récupère toutes les machines compatibles avec un traitement donné en fonction de la localisation et de la taille de la tumeur.
  - **Fonctionnement :** En fonction de l'ID du centre et des caractéristiques du traitement (localisation et taille), retourne une liste des machines compatibles pour traiter cette localisation et les dimensions requises.
- `GetAddedPatients()` :
  - **But :** Retourne la liste des patients ajoutés dans le système.
  - **Fonctionnement :** Renvoie directement la liste des patients stockée dans le service `patientListService`.
- `GetSchedulePatient()` :
  - **But :** Permet de récupérer le planning d'un patient spécifique.
  - **Fonctionnement :** Prend le nom du patient et retourne son planning associé en fonction de son nom.
- `AddPatientAPI()` :
  - **But :** Ajoute un patient et tente d'automatiser la planification des traitements en ajoutant des horaires automatiquement.
  - **Fonctionnement :** Prend plusieurs paramètres (nom, prénom, machine, localisation, etc.) et tente d'ajouter un patient au système. Vérifie la disponibilité des créneaux et ajoute le patient à la liste si l'ajout réussit.
- `DeletePatient()` :
  - **But :** Supprime un patient de la liste des patients ajoutés.
  - **Fonctionnement :** Prend en entrée le nom du patient à supprimer, trouve ce

patient dans la liste et le retire.

- `updateEvent()` :
  - **But :** Met à jour les événements du planning.
  - **Fonctionnement :** Reçoit une liste d'événements modifiés et met à jour les horaires des événements correspondants pour les patients dans la base de données.
- `AddSchedule()` :
  - **But :** Ajoute manuellement un horaire à un patient.
  - **Fonctionnement :** Prend en entrée le nom du patient, la date de début, la date de fin et la machine utilisée pour ajouter un nouveau créneau de traitement pour ce patient.
- `RemoveSchedule()` :
  - **But :** Supprime un horaire spécifique pour un patient.
  - **Fonctionnement :** Supprime un horaire de traitement pour un patient en fonction de l'ID du planning à supprimer. Si le patient ou l'horaire n'est pas trouvé, une erreur est renvoyée.
- `UpdateSchedule()` :
  - **But :** Met à jour un événement particulier du planning d'un patient, comme le centre et la machine.
  - **Fonctionnement :** Permet de modifier un événement spécifique de traitement en ajustant des informations comme la machine et le centre de traitement, selon les besoins.

4.3 StatsController : Gestion des Statistiques

Le **StatsController** fournit des statistiques et des rapports basés sur les données disponibles, aidant à la prise de décision et à l'analyse des performances.

Le contrôleur `StatsController` gère les statistiques liées aux machines et aux rendez-vous dans l'application MediPlanner. Il inclut des méthodes pour récupérer les informations sur les rendez-vous du jour ainsi que les maintenances des machines.

1. **Page de bienvenue :**
   — `HelloPage` : Renvoie un message de bienvenue "hello" pour vérifier que l'API fonctionne correctement.
2. **Nombre de patients pour aujourd'hui :**
   — `GetNumberPatient` : Calcule et renvoie le nombre total de patients ayant un rendez-vous pour aujourd'hui. Il itère sur toutes les machines du centre et utilise la méthode `GetMachineNumberPatient` pour chaque machine.
3. **Maintenance des machines :**
   — `GetMachinesMaintenance` : Récupère toutes les maintenances planifiées pour les machines du centre pour la journée actuelle. Si une machine est en maintenance, cette maintenance est ajoutée à la liste des résultats.

4.3.1 Points clés

— Le contrôleur utilise le contexte `VarianContext` pour accéder à la base de données et interagir avec les données des machines et des rendez-vous.
— Les méthodes `GetNumberPatient` et `GetMachinesMaintenance` itèrent sur toutes les machines du centre, en utilisant les requêtes de la classe `MachineScheduleQuery` pour obtenir les données nécessaires.
— Le contrôleur fournit des informations utiles sur les rendez-vous du jour ainsi que sur les maintenances des machines, ce qui est essentiel pour la gestion opérationnelle des centres.

4.4 Les services

4.5 CenterService

Le code suivant gère le service des centres dans le projet **MediPlanner**. Ce service permet d'ajouter, de supprimer, de modifier et de récupérer des centres, ainsi que de gérer les ressources humaines et matérielles associées à chaque centre.

4.5.1 Méthodes principales

- **AddCenter** : Cette méthode permet d'ajouter un centre à la liste des centres.
  - Elle prend un objet `Center` en paramètre.
  - Après avoir ajouté ce centre à la liste des centres, elle retourne `true` pour indiquer que l'ajout a été effectué avec succès.
- **DeleteCenter** : Cette méthode permet de supprimer un centre de la liste.
  - Elle prend un objet `Center?` en paramètre.
  - Si le centre est nul, la méthode retourne `false`.
  - Si le centre existe, il est retiré de la liste des centres et la méthode retourne `true`.
- **GetCenter** : Cette méthode permet de récupérer un centre à partir de son identifiant.
  - Elle prend un `id` en paramètre.
  - Elle utilise la méthode `FirstOrDefault` pour rechercher le centre dont l'identifiant correspond à l'`id`.
  - Si aucun centre n'est trouvé, elle retourne `null`.
- **EditCenterRessources** : Cette méthode permet de modifier les ressources humaines associées à un centre.
  - Elle prend un objet `Center` et une liste de `Ressources` en paramètres.
  - La méthode remplace les ressources existantes du centre par celles fournies dans la liste.
  - Elle retourne `true` pour indiquer que l'opération a été réussie.
- **EditCenterMachines** : Cette méthode permet de modifier les machines associées à un centre.
  - Elle prend un objet `Center` et une liste de `Machine` en paramètres.
  - Pour chaque machine, elle réinitialise les localisations disponibles et assigne un paramètre par défaut.
  - Les machines sont ensuite affectées au centre et la méthode retourne `true` pour confirmer la modification.

4.5.2 Résumé des fonctionnalités clés

- **Gestion des centres** : Ajout, suppression et récupération des centres.
- **Modification des ressources humaines** : Modification des ressources humaines d'un centre en remplaçant celles existantes.
- **Modification des machines** : Modification des machines d'un centre, avec réinitialisation des localisations et des paramètres associés.

4.5.3 Améliorations possibles

- **Optimisation de la gestion des ressources** : Ajouter une gestion plus fine des ressources pour permettre des ajouts ou des suppressions sans écrasement complet.
- **Validation des machines** : Ajouter des mécanismes de validation pour s'assurer que les machines sont valides avant de les ajouter au centre.
- **Amélioration des performances** : Mettre en place des mécanismes de mise en cache pour améliorer la performance lors de la récupération ou de la modification de centres.

Ce code permet de gérer efficacement les centres, leurs ressources humaines et matérielles dans le système de **MediPlanner**, en offrant une interface simple pour effectuer des ajouts, suppressions et modifications.

4.6 PatientListService

Le code suivant gère le service de la liste des patients dans le projet **MediPlanner**. Ce service permet de stocker et d'accéder aux patients dans une collection.

4.6.1 PatientListService

- **patients** : Une propriété qui contient la liste des patients. C'est une liste de type `List<Patient>`.
  - Cette liste est initialisée vide dans le constructeur de la classe.
- **Constructeur** : Le constructeur `PatientListService` initialise la propriété `patients` en tant que nouvelle liste vide de patients.
  - Il n'y a pas de paramètres d'entrée.
  - Cela permet de commencer avec une liste vide de patients.

4.6.2 Classe Patient

La classe `Patient` est responsable de la gestion des informations relatives à un patient, telles que son nom et la liste des événements qui le concernent.

- **Propriétés**
  - `FirstName` : Le prénom du patient.
  - `LastName` : Le nom de famille du patient.
  - `Schedules` : La liste des événements planifiés pour ce patient.
- **Constructeurs**
  - Le constructeur par défaut initialise `FirstName` et `LastName` à des chaînes vides et `Schedules` à une liste vide.
  - Le second constructeur prend en entrée `firstName` et `lastName` et initialise `Schedules` à une liste vide.
  - Le troisième constructeur prend en entrée `firstName`, `lastName` et `schedules` pour initialiser les propriétés correspondantes.
- **Méthode ToString**
  - La méthode `ToString()` retourne une représentation sous forme de chaîne du patient, sous le format `FirstName LastName`.

4.6.3 Résumé des fonctionnalités clés

— **Gestion des patients** : Permet de stocker une liste de patients, avec des informations telles que le prénom, le nom et les événements associés.
— **Gestion des événements planifiés** : Chaque patient possède une liste d'événements (`Schedules`) à partir de la classe `ScheduleQueryResult`.

4.6.4 Améliorations possibles

— **Gestion des événements** : Ajouter des méthodes permettant de modifier la liste des événements d'un patient (ajouter/supprimer des événements).
— **Validation des données** : Implémenter des mécanismes de validation pour s'assurer que les informations des patients sont valides avant de les ajouter à la liste.
— **Optimisation de la recherche** : Ajouter des fonctionnalités pour rechercher des patients dans la liste par nom, prénom, ou événements.

4.6.5 Résumé

Ce code permet de gérer une liste de patients, stockant des informations comme le prénom, le nom, et les événements associés. La classe `PatientListService` fournit une structure pour gérer cette liste, et la classe `Patient` contient les informations détaillées pour chaque patient.

4.6.6 Interaction des services

4.7 Services utilisés dans le contexte global

Le `PatientListService` est responsable de la gestion des patients et de leurs créneaux associés. Lorsqu'un patient est ajouté, ses créneaux sont calculés et stockés dans la liste des patients. Cette liste permet de retrouver le patient et de conserver son état, ou de le modifier ultérieurement si nécessaire. Ce service travaille en collaboration avec le `PlanningController`, qui gère les opérations suivantes :

— **Ajout de créneaux** : Lorsqu'un patient est ajouté, des créneaux sont calculés pour ce dernier, puis ajoutés à son profil.
— **Suppression de créneaux** : Si des créneaux doivent être supprimés, le `PlanningController` interagira avec le `PatientListService` pour supprimer ces créneaux de la liste.
— **Mise à jour de créneaux** : Si un patient doit voir ses créneaux mis à jour, ces informations seront modifiées dans la liste.

En résumé, `PatientListService` stocke les informations du patient et de ses créneaux, facilitant ainsi leur gestion tout au long du processus de traitement.

Le `CenterListService` est utilisé dans le `AdminController` pour gérer les centres. Il permet d'ajouter des centres, de les modifier et de mettre à jour leurs ressources associées. Ce service inclut la gestion des ressources humaines et matérielles dans chaque centre, comme les machines et le personnel.

Voici les principales fonctions du `CenterListService` :

— **Ajout de centres** : Ce service permet d'ajouter de nouveaux centres à la liste, en fournissant les informations nécessaires, telles que le nom et la localisation.
— **Modification de centres** : Il est possible de modifier les centres existants, par exemple en changeant leurs ressources ou leur nom.

— **Assignation de machines et de ressources humaines** : Le service permet d'assigner des machines et des ressources humaines à chaque centre, ce qui est essentiel pour le bon fonctionnement du centre.

L'interaction entre ces services se produit dans des scénarios où un patient peut être affecté à un centre spécifique. Par exemple, lorsqu'un patient est ajouté, il pourrait être affecté à un centre particulier et se voir assigner des machines et des ressources humaines en fonction des disponibilités de ce centre.

5 MediData : La base de données

MediData est la couche du projet `MediCore` qui gère l'accès aux données via `Entity Framework Core`. Cette couche utilise le package NuGet `Microsoft.EntityFrameworkCore` pour interroger la base de données et extraire les informations nécessaires, sans se soucier de l'ajout, de la mise à jour ou de la suppression de données.

5.1 Contexte de données (`DbContext`)

Le `DbContext` est la classe centrale utilisée pour interagir avec la base de données. Elle représente une session de travail avec la base de données et permet d'effectuer des requêtes pour extraire des données. Dans MediData, un `DbContext` est utilisé pour définir l'accès aux entités du projet et exécuter des requêtes.

Exemple de `DbContext` :

```
public class MediDataContext : DbContext
{
    public DbSet<Patient> Patients { get; set; }
    public DbSet<Machine> Machines { get; set; }
    // Autres DbSet pour les autres entités
}
```

`DbSet` représente une collection d'entités qui peuvent être interrogées pour extraire les données de la base de données.

5.2 Modèles d'entités

Les modèles d'entités sont des classes qui représentent les tables de la base de données. Chaque propriété d'une classe représente une colonne dans la table correspondante. Ces modèles sont utilisés pour effectuer des requêtes afin de récupérer les données stockées dans la base.

Exemple d'un modèle d'entité `Patient` :

```
public class Patient
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Location { get; set; }
    public DateTime PreferredDate { get; set; }
}
```

5.3 Extraction des données

Une fois les entités et le `DbContext` définis, il est possible de récupérer des données en exécutant des requêtes sur les collections `DbSet`. Ces requêtes peuvent être simples ou complexes, en fonction des besoins.

Exemple de récupération d'un patient :

```
using (var context = new MediDataContext())
{
    var patient = context.Patients.FirstOrDefault(p => p.Id == 1);
}
```

Dans cet exemple, nous interrogeons la base de données pour récupérer le patient ayant l'ID 1. La méthode `FirstOrDefault()` renvoie le premier patient trouvé ou `null` si aucun patient n'est trouvé.

Exemple de récupération de tous les patients :

```
using (var context = new MediDataContext())
{
    var patients = context.Patients.ToList();
}
```

Ici, nous récupérons tous les patients présents dans la base de données en appelant `ToList()` sur le `DbSet` des patients.

5.4 Filtrage et Tri des Données

`Entity Framework Core` permet également de filtrer, trier et paginer les résultats des requêtes. Par exemple, vous pouvez récupérer tous les patients d'une certaine localisation ou trier les patients par nom de famille.

Exemple de filtrage par localisation :

```
using (var context = new MediDataContext())
{
    var patientsInLocation = context.Patients
                                    .Where(p => p.Location == "Paris")
                                    .ToList();
}
```

Exemple de tri des patients par nom :

```
using (var context = new MediDataContext())
{
    var sortedPatients = context.Patients
                                .OrderBy(p => p.LastName)
                                .ToList();
}
```

Ces méthodes permettent de personnaliser les requêtes et d'extraire des données spécifiques en fonction des critères définis.

5.5 Joins et GroupJoin

Les jointures permettent de lier des données provenant de plusieurs tables. `Entity Framework Core` permet d'utiliser les jointures avec la méthode `Join`. Cette méthode permet de joindre deux collections en fonction d'un critère spécifique, et de récupérer un ensemble combiné de données.

Exemple de jointure entre les tables `Patients` et `Machines` :

```
using (var context = new MediDataContext())
{
    var patientMachineJoin = context.Patients
        .Join(context.Machines,
              patient => patient.Id,
              machine => machine.PatientId,
              (patient, machine) => new
              {
                  PatientName = patient.FirstName + " " + patient.LastName,
                  MachineName = machine.Name
              })
        .ToList();
}
```

Dans cet exemple, nous effectuons une jointure entre la table des patients et la table des machines en utilisant l'ID du patient, puis nous sélectionnons le nom du patient et le nom de la machine associée.

Les jointures groupées (ou `GroupJoin`) permettent de créer des groupes d'éléments basés sur une clé de jointure. Cela permet d'obtenir une liste d'éléments associés à un seul élément principal.

Exemple de jointure groupée entre `Patients` et `Machines` :

```
using (var context = new MediDataContext())
{
    var patientWithMachines = context.Patients
         .GroupJoin(context.Machines,
                    patient => patient.Id,
                    machine => machine.PatientId,
                    (patient, machines) => new
                    {
                        PatientName = patient.FirstName + " " + patient.LastName,
                        Machines = machines
                    })
         .ToList();
}
```

Ici, pour chaque patient, nous regroupons les machines associées dans un groupe. Le résultat est une liste de patients avec leurs machines respectives, permettant d'obtenir toutes les machines qui sont associées à chaque patient.

5.6 Résumé

En résumé, `MediData` est la couche du projet `MediCore` responsable de l'accès aux données à l'aide d'Entity Framework Core. Elle permet de définir les entités qui représentent les tables de la base de données et d'exécuter des requêtes pour extraire les données nécessaires au projet. Ces opérations sont réalisées de manière flexible et puissante, en utilisant les fonctionnalités offertes par `Entity Framework Core` pour filtrer, trier et manipuler les données extraites de la base de données.

6 MediPlanner : Partie Logique

La partie **MediPlanner** de l'application gère la logique interne de la planification des traitements, des patients, des machines et des événements associés. Cette couche est responsable de l'intégration et du traitement des données provenant des autres sous-projets, à savoir **MediCore** (qui gère la structure de l'API) et **MediData** (qui fournit l'accès aux données via Entity Framework).

MediPlanner assure la gestion des horaires de traitement, l'optimisation de l'utilisation des machines et l'automatisation de la planification des événements en fonction des disponibilités des machines et des besoins des patients.

6.1 Les requêtes

6.2 MachineScheduleQueries

Le code présenté gère les requêtes liées au planning des machines dans le projet **MediPlanner**. Voici une explication de son fonctionnement, ainsi qu'un aperçu des méthodes clés.

6.2.1 Constantes

- **MIN_FREE_TIME** : Temps minimum de 10 minutes de temps libre entre les activités.
- **MACHINE_START_HOUR et MACHINE_END_HOUR** : Les heures de début et de fin de la machine pour chaque journée.

6.2.2 Méthodes principales

- **GetActivityOnADay** : Cette méthode récupère les activités d'une machine pour un jour spécifique. Elle utilise la méthode `QueryMachineActivities` pour récupérer les activités de cette journée et ajoute les plages horaires libres avec la méthode `FillFreeSlots`.
- **GetScheduleActivity** : Cette méthode récupère les activités d'une machine sur une plage de plusieurs semaines. Elle :
  - Récupère les activités via `QueryMachineActivities`.
  - Ajoute les vacances avec `GetHolidays`.
  - Ajoute les informations des patients qui utilisent cette machine.
  - Ajoute des signaux pour chaque jour (début et fin de journée).
  - Les activités sont ensuite regroupées par jour avec `GroupActivitiesByDay`.
  - Les plages libres sont insérées avec `FillFreeSlots`.
- **QueryMachineActivities** : Cette méthode est le cœur de la récupération des données. Elle exécute une série de jointures entre les différentes entités de la base de données pour récupérer les activités d'une machine entre deux dates. Les résultats sont filtrés pour ne prendre que les activités non supprimées et dans l'intervalle des dates données. Elle renvoie les résultats sous la forme d'une liste de `ScheduleQueryResult`, qui inclut des détails sur l'activité (heure de début et de fin, statut, etc.).
- **GetHolidays** : Cette méthode récupère les vacances planifiées pour la période donnée. Les vacances sont ajoutées à la liste des activités sous forme de *HOLIDAY*.

— **AddSignals** : Cette méthode ajoute des *signaux* à des heures spécifiques pour chaque jour, marqués comme *SIGNAL*. Cela permet de marquer le début et la fin des journées de traitement.
— **GroupActivitiesByDay** : Cette méthode regroupe les activités par jour (date). Elle trie les activités par heure de début, puis les regroupe dans des listes de jour, en fonction de la date.
— **FillFreeSlots** : Cette méthode remplit les plages horaires libres entre les activités si elles sont suffisamment grandes (minimum 10 minutes). Cela permet d'afficher des créneaux horaires vides entre deux traitements ou entre un traitement et une maintenance.
— **GetMachineUsedTime** : Cette méthode récupère le temps total d'utilisation d'une machine sur une période donnée. Cela est calculé en prenant la somme des différences entre l'heure de début et de fin de chaque activité.
— **GetMachineNumberPatient** : Cette méthode récupère le nombre de patients qui utilisent la machine à une date spécifique.
— **GetMaintenance** : Cette méthode récupère les informations sur les maintenances programmées pour une machine à une date donnée. La méthode vérifie si des maintenances sont prévues ou en cours sur cette machine.

6.2.3 Résumé des fonctionnalités clés

— **Planification des activités** : Le code permet de gérer les horaires des traitements, les maintenances, et d'optimiser les créneaux disponibles.
— **Gestion des vacances et des signaux** : Les vacances sont intégrées dans les plannings, et les signaux marquent les périodes de début et de fin de chaque journée.
— **Génération d'activités libres** : Si des plages horaires sont disponibles entre deux activités, elles sont ajoutées à la liste pour être visibles.
— **Suivi du temps d'utilisation des machines** : Le temps total d'utilisation d'une machine est calculé, ainsi que le nombre de patients planifiés pour une machine donnée.

6.2.4 Améliorations possibles

— **Optimisation des requêtes** : Certaines requêtes pourraient être optimisées en ajoutant des index ou en utilisant des méthodes d'accès plus efficaces pour gérer de grandes quantités de données.
— **Gestion des erreurs** : Ajout de gestion d'exceptions dans les méthodes de récupération de données (par exemple, gestion des erreurs de connexion à la base de données).
— **Sécurité** : Vérification de l'accès aux données sensibles pour s'assurer que seules les personnes autorisées peuvent accéder aux informations sur les patients ou les plannings.

Ce code permet de récupérer, organiser et afficher de manière efficace les horaires des machines et des patients tout en prenant en compte les maintenances et les plages horaires disponibles.

6.3 AdminQueries

Le code suivant gère les requêtes pour le contrôleur d'administration dans le projet **MediPlanner**. Il permet de récupérer les machines et les ressources, ainsi que d'effectuer des recherches basées sur des identifiants spécifiques.

6.3.1 Méthodes principales

- **GetMachines** : Cette méthode récupère une liste de machines actives, triées par nom. Elle exclut les machines dont le statut est *Disabled* ou *Virtual*. Les machines retournées contiennent uniquement leur *Id* et *Name*.
  - Elle effectue une requête LINQ sur le contexte `VarianContext` pour récupérer les machines valides.
  - La liste est ensuite triée par nom de machine et les machines sont retournées sous forme d'une liste de `Class.Machine`.
- **GetRessources** : Cette méthode récupère toutes les ressources humaines (membres du personnel) disponibles. Elle retourne une liste contenant l'identifiant, le prénom et le nom de chaque membre du personnel.
  - Elle exécute une requête LINQ pour obtenir tous les membres du personnel (`Staff`) et les trier par prénom.
  - Les résultats sont projetés dans une liste de `Ressources` contenant uniquement les informations nécessaires.
- **GetRessourcesById** : Cette méthode récupère des ressources spécifiques basées sur une liste d'identifiants. Elle filtre les ressources pour ne retourner que celles dont les identifiants sont dans la liste `Ids`.
  - Comme pour `GetRessources`, une requête LINQ est utilisée pour récupérer toutes les ressources disponibles.
  - La liste est ensuite filtrée en fonction des identifiants présents dans `Ids`.
- **GetMachinesById** : Cette méthode fonctionne de manière similaire à `GetRessourcesById`, mais elle récupère les machines basées sur une liste d'identifiants.
  - Elle commence par récupérer toutes les machines disponibles.
  - Ensuite, elle filtre la liste pour ne conserver que celles dont l'identifiant figure dans la liste `Ids`.

6.3.2 Résumé des fonctionnalités clés

- **Gestion des machines** : Récupération des machines actives, avec filtrage selon leur statut.
- **Gestion des ressources humaines** : Récupération de toutes les ressources disponibles, ou filtrage basé sur des identifiants donnés.
- **Requêtes filtrées par identifiant** : Recherche ciblée de machines ou de ressources en fonction d'une liste d'identifiants spécifiques.

6.3.3 Améliorations possibles

- **Optimisation des requêtes** : Ajouter des filtres supplémentaires ou utiliser des jointures pour limiter la quantité de données renvoyées.
- **Gestion des erreurs** : Ajouter des mécanismes de gestion d'erreurs pour traiter les éventuelles erreurs de base de données ou les identifiants invalides.

— **Améliorer la pagination** : Implémenter la pagination pour gérer de grandes quantités de ressources ou de machines afin d'améliorer la performance des requêtes.

Ce code permet de récupérer et de filtrer les informations des machines et des ressources humaines en fonction des besoins de l'administrateur dans **MediPlanner** afin des les assigner aux différents sites.

7 Partie Front : Python et HTML

7.1 Concept de Django

Django est un framework web en Python qui permet de développer des applications web rapidement et de manière efficace. Il suit le principe **MTV** (Model-Template-View), qui est une variation du traditionnel modèle **MVC** (Model-View-Controller). Dans le cadre de Django, les principaux composants sont les suivants :

7.1.1 Les Vues (Views)

Les **vues** dans Django sont des fonctions ou des classes qui reçoivent des requêtes HTTP et retournent des réponses HTTP. Une vue peut interagir avec des modèles pour récupérer des données, exécuter des logiques métier, et rendre des templates HTML. En fonction de la demande de l'utilisateur, une vue peut afficher une page, rediriger vers une autre page, ou effectuer d'autres actions.

Une vue en Django peut être définie comme une fonction simple ou une classe qui hérite de `View`. Par exemple :

```
from django.http import HttpResponse
from django.shortcuts import render

def home_view(request):
    return HttpResponse('Bienvenue à ClinicApp!')
```

Ici, la vue `home_view` prend une requête HTTP en entrée et retourne une réponse HTTP, qui est une simple chaîne de texte dans ce cas.

Les vues peuvent aussi être plus complexes, en rendant des templates HTML qui sont remplis avec des données dynamiques issues des modèles.

7.1.2 Les Formulaires (Forms)

Les **formulaires** dans Django sont utilisés pour recueillir des informations des utilisateurs. Django fournit une bibliothèque `forms` qui permet de créer, valider et manipuler des formulaires HTML de manière sécurisée. Les formulaires peuvent être utilisés pour des tâches telles que l'inscription d'un patient, la mise à jour de données ou la soumission de requêtes de traitement.

Voici un exemple simple de formulaire Django pour un patient :

```
from django import forms

class PatientForm(forms.Form):
    name = forms.CharField(label='Nom', max_length=100)
    email = forms.EmailField(label='Email')
    date_of_birth = forms.DateField(label='Date de Naissance')
```

Ensuite, dans une vue, on pourrait utiliser ce formulaire pour afficher ou traiter les données soumises par un utilisateur :

```
def patient_view(request):
    if request.method == 'POST':
        form = PatientForm(request.POST)
        if form.is_valid():
            # Traitement des données du formulaire
            return HttpResponse('Données du patient soumises!')
    else:
        form = PatientForm()
    return render(request, 'patient_form.html', {'form': form})
```

Ici, `PatientForm` est un formulaire qui recueille le nom, l'email et la date de naissance d'un patient. La vue `patient_view` gère l'affichage du formulaire et le traitement des données soumises.

7.1.3 Les Fichiers Statics (Static Files)

Les fichiers statiques dans Django sont des fichiers tels que des feuilles de style CSS, des fichiers JavaScript et des images qui sont utilisés dans le front-end de l'application. Ces fichiers ne changent pas en fonction des actions de l'utilisateur et sont servis directement par le serveur web.

Django gère les fichiers statiques grâce au dossier `static`. Le framework permet d'organiser et de servir ces fichiers dans les templates HTML à l'aide de la balise `static`. Par exemple :

```
{% load static %}
<img src="{% static 'images/logo.png' %}" alt="Logo">
<link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">
```

Ici, l'image et la feuille de style sont référencées à partir du dossier `static` et Django gère leur chemin d'accès pour les servir correctement lors du rendu du template HTML.

7.1.4 Conclusion

En résumé, Django permet une gestion efficace des vues, des formulaires et des fichiers statiques. Les vues traitent la logique métier et les interactions avec les utilisateurs, les formulaires offrent une interface pour collecter des données, et les fichiers statiques permettent d'enrichir l'expérience utilisateur avec des éléments visuels et interactifs. Django simplifie le processus de développement web en séparant ces responsabilités tout en offrant des outils puissants pour créer des applications robustes.

7.1.5 Explication des URLs

Les URLs dans une application Django sont définies dans le fichier `urls.py`. Elles permettent de relier les requêtes HTTP à des vues spécifiques dans l'application. Ce système est basé sur le principe de "route matching", où chaque URL est associée à une vue qui gère la logique de la requête. Le fichier `urls.py` contient également des paramètres de nom pour chaque chemin, ce qui permet de référencer les URLs dans les templates et d'autres parties de l'application.

Voici un aperçu de la configuration des URLs de l'application :

- **Accueil et Feedback**
  - `''` : Accède à la vue `home`, qui représente la page d'accueil de l'application.
  - `'feedback/'` : Dirige vers la vue `feedback_view`, où les utilisateurs peuvent soumettre des retours. (ne sert à rien pour le moment)
- **Gestion des Patients**
  - `'add_patient/'` : Permet d'ajouter un nouveau patient via la vue `add_patient`.
  - `'choose_machine/'` : Dirige vers la vue `choose_machine` où l'utilisateur peut sélectionner une machine pour un patient.
  - `'deletePatient'` : Supprime un patient en envoyant une requête à la vue `deletepatient`.
  - `'get_patient_schedules'` : Récupère les créneaux de planning pour un patient via la vue `fetchSchedulePatient`.
- **Gestion du Planning**
  - `'view_schedule/'` : Affiche le planning des machines via la vue `view_schedule`.
  - `'fetchEventsChanges'` : Récupère les modifications d'événements via la vue `fetchEventsChanges`.
  - `'updateEvents'` : Permet de mettre à jour les événements dans le planning via la vue `updateEvents`.
  - `'addSchedule'` : Permet d'ajouter un créneau de planning via la vue `addSchedule`.
  - `'deleteSchedule'` : Supprime un créneau de planning via la vue `removeSchedule`.
  - `'updateSchedule'` : Permet de mettre à jour un créneau de planning via la vue `updateSchedule`.
- **Gestion des Centres (Administration)**
  - `'admin/centres/'` : Affiche la liste des centres via la vue `list_centres`.
  - `'admin/centres/create/'` : Permet de créer un centre via la vue `create_centre`.
  - `'admin/centres/edit/<int:centre_id>/'` : Permet de modifier un centre spécifique via la vue `edit_centre`, avec un identifiant de centre (`centre_id`).
  - `'admin/editMachinePreferences/<int:centre_id>/<str:machine_name>/'` : Permet de modifier les préférences d'une machine dans un centre spécifique via la vue `edit_machine_preferences`.
- **Importation et Exportation de Configurations de Centres**
  - `'import_centre_config/'` : Permet d'importer la configuration d'un centre via la vue `import_centre_config`.
  - `'export_centre/'` : Permet d'exporter la configuration d'un centre via la vue `export_centre`.
- **Admin**
  - `'admin/'` : Accède à la page d'administration de Django via `admin.site.urls`.

7.2 Les vues du projet

Les vues de ce projet Django sont conçues pour gérer l'interaction avec l'utilisateur en traitant diverses opérations liées à la gestion des patients, des machines, des plannings et des centres de traitement. Chaque vue a un rôle spécifique et permet de réaliser des actions telles que l'ajout de patients, la sélection de machines, la gestion des plannings ou encore l'administration des centres.

7.2.1 Vue : Home

La vue `home` est responsable de la page d'accueil de l'application. Elle efface les sessions en cours et récupère les données liées aux machines via l'API. Ces données incluent :

— Le statut des machines
— Les machines en maintenance
— Le nombre de patients

Elle rend ensuite ces données sur la page d'accueil, en passant ces informations au template `patients/home.html`.

7.2.2 Vue : Add Patient

La vue `add_patient` permet d'ajouter un patient. Lorsqu'une requête `POST` est envoyée, les informations du formulaire sont récupérées, telles que :

— L'urgence du patient
— Le nombre de séances par semaine
— La durée du traitement (en semaines)
— Les données personnelles du patient (nom, prénom, localisation, etc.)
— L'heure de traitement (si nécessaire)

Une fois les données validées, elles sont envoyées dans la requête pour rediriger l'utilisateur vers la page de choix de machine (`choose_machine`), où les informations du patient seront utilisées pour sélectionner la machine.

7.2.3 Vue : Choose Machine

La vue `choose_machine` permet à l'utilisateur de choisir une machine pour le traitement du patient. Cette vue reçoit des paramètres via `GET`, tels que :

— L'urgence du patient
— La localisation du traitement
— Les détails du patient (nom, prénom, etc.)

Elle récupère également les centres de traitement disponibles via l'API et affiche les machines compatibles avec les spécifications du patient. Si une machine est sélectionnée, une redirection est effectuée vers la vue `view_schedule` pour afficher le planning de traitement.

7.2.4 Vue : View Schedule

La vue `view_schedule` affiche le planning des traitements pour un patient donné. Elle récupère les paramètres nécessaires tels que la machine sélectionnée et le centre de traitement, puis charge les centres via l'API. Le formulaire d'ajout de patient est également affiché dans cette vue, permettant d'ajouter de nouveaux patients tout en consultant le planning existant.

7.2.5 Vue : Get Machines for Centre

Cette vue permet de récupérer les machines disponibles pour un centre spécifique. Lorsque l'ID du centre est passé via `GET`, elle interroge l'API pour obtenir les machines de ce centre et retourne les résultats sous forme de réponse JSON.

7.2.6 Vue : Feedback View

La vue `feedback_view` permet aux utilisateurs de soumettre des retours via un formulaire. Si une requête `POST` est envoyée avec des données valides, ces dernières sont enregistrées et l'utilisateur est redirigé vers la page d'accueil.

7.2.7 Vues : Centres

- **List Centres**
  - Récupère la liste des centres de traitement via l'API.
  - Affiche la liste des centres sur une page.
- **Create Centre**
  - Permet de créer un nouveau centre de traitement.
  - Si une requête `POST` est envoyée avec des données valides, un nouveau centre est créé via l'API.
- **Edit Centre**
  - Permet de modifier les ressources et machines associées à un centre existant.
  - Récupère les données du centre et les ressources disponibles via l'API.
  - Permet à l'utilisateur de mettre à jour les ressources et machines associées.

7.2.8 Vues : Gestion des Événements

- **Update Schedule**
  - Permet de mettre à jour un créneau de planning existant.
  - Si une requête `POST` est envoyée avec des informations sur l'événement, celles-ci sont mises à jour dans la base de données via l'API.
- **Add Schedule**
  - Permet d'ajouter un nouveau créneau de planning.
  - Les données du créneau (nom, heure de début, heure de fin, machine) sont envoyées à l'API pour être ajoutées au planning.
- **Remove Schedule**
  - Permet de supprimer un créneau de planning.
  - Envoie l'ID du créneau à supprimer via une requête `POST`.

7.3 Les interactions avec l'API RESTful C# MediPlanner

Dans ce projet, le backend Python, qui utilise le framework Django, interagit avec une API RESTful développée en C# pour effectuer diverses opérations liées au planning des machines, à la gestion des patients et des centres. L'API RESTful fournit des endpoints qui permettent de récupérer, ajouter, modifier et supprimer des informations dans le système de gestion de planning. Ces interactions sont principalement réalisées à l'aide de requêtes HTTP (GET, POST, DELETE) envoyées depuis le serveur Python vers l'API.

L'objectif principal de cette interaction est de centraliser la gestion des plannings, des machines et des patients via un point d'accès unique (l'API RESTful). Le backend Django envoie des requêtes à l'API pour obtenir les informations nécessaires, les traiter et les afficher dans l'interface utilisateur. Cela permet d'assurer la synchronisation des données entre l'application Python et le système MediPlanner.

- **fetch_machine_data** : Cette fonction interroge l'API pour obtenir des informations sur l'état des machines, comme le nombre de patients ou la liste des machines en maintenance. Elle fait deux requêtes GET pour obtenir ces informations et les formate avant de les renvoyer.
- **fetch_machines_choices** : Cette fonction permet de récupérer la machine préférée et les machines compatibles pour un patient donné. Elle utilise les paramètres du formulaire (comme le centre de traitement et la machine sélectionnée) pour effectuer des requêtes GET à l'API et récupérer les informations nécessaires.
- **get_form_view_param** : Cette fonction extrait les paramètres du formulaire de la requête HTTP, tels que la machine sélectionnée et le centre choisi par l'utilisateur, pour les utiliser dans d'autres fonctions.
- **get_centre_details** : Cette fonction récupère les détails d'un centre spécifique en utilisant son identifiant. Elle effectue une requête GET pour obtenir les informations du centre, et renvoie un objet JSON contenant ces données.
- **get_centres_from_api** : Cette fonction interroge l'API pour obtenir la liste de tous les centres disponibles. Elle effectue une requête GET à l'API et renvoie un tableau d'objets JSON représentant les centres.
- **fetch_schedule_from_api** : Cette fonction récupère le planning de la machine pour une semaine donnée. Elle effectue une requête GET en passant la machine sélectionnée et la date de début de la semaine comme paramètres.
- **fetch_patient_schedule_from_api** : Cette fonction permet d'ajouter un patient au planning. Elle envoie une requête GET avec plusieurs paramètres (nom du patient, taille de la tumeur, heure de début, etc.) pour ajouter le patient à l'API.
- **send_update_event** : Cette fonction envoie des informations sur des événements modifiés à l'API. Elle transforme les événements en dictionnaires, puis envoie ces données sous forme de JSON dans une requête POST à l'API.
- **send_add_schedule** : Cette fonction permet d'ajouter un événement de planning à l'API. Elle envoie les informations de l'événement (nom, heure de début et de fin, machine) dans une requête POST.
- **send_update_schedule** : Cette fonction met à jour un événement de planning existant dans l'API. Elle envoie les informations mises à jour dans une requête POST.
- **send_remove_schedule** : Cette fonction permet de supprimer un événement de planning. Elle envoie une requête DELETE à l'API avec l'ID de l'événement à

supprimer.

— **getAddedPatient** : Cette fonction récupère la liste des patients ajoutés via l'API. Elle effectue une requête GET pour obtenir les informations de tous les patients qui ont été ajoutés au système.
— **send_remove_patient** : Cette fonction permet de supprimer un patient de l'API en envoyant une requête DELETE avec le nom du patient à supprimer.
— **send_import_file** : Cette fonction permet d'importer un fichier contenant des informations sur les centres. Elle envoie les données sous forme de JSON dans une requête POST à l'API.
— **send_machine_pref** : Cette fonction permet de mettre à jour les préférences de machines pour un centre donné. Elle envoie une requête POST à l'API avec les données de préférence de la machine.
— **getSchedulePatient** : Cette fonction récupère le planning d'un patient en particulier. Elle envoie une requête GET avec le nom du patient et récupère le planning sous forme de JSON.