Crud

CRUD est un mnémonique qui représente les opérations de base en lien avec la gestion d'informations numériques.

> ***C** : Create (Créer / Ajouter)*

> ***R** : Read (Lire / Charger)*

> ***U** : Update (Modifier)*

> ***D** : Delete (Supprimer)*

Create

Utilisateur

Pour commencer, on aimerait ajouter une information.

Pour demander et récupérer une information donnée par l'utilisateur, par exemple, un nom de paquet de cartes, on a besoin soit

- D'un composant `Entry`

soit

- De la fonction `DisplayPromptAsync`

Exemple de code avec DisplayPromptAsync

```
string newName = await DisplayPromptAsync("Renommer", "Nouveau nom du paquet:",
    "Enregistrer", "Annuler", initialValue: currentDeck.Name);

if (string.IsNullOrWhiteSpace(newName))
    return;

currentDeck.Name = newName;
```

Stockage

Une fois qu'on récupère, par exemple, le nom d'un paquet sous forme de `string`, on doit choisir quoi en faire. Une option serait d'utiliser une base de données, mais cela implique une certaine complexité et on peut déjà réaliser une version simplifiée avec un système de `persistance` basé sur des fichiers.

JSON

Un format facile à gérer pour l'ordinateur et relativement digeste pour un humain est le format JSON. Ce dernier a l'avantage d'être pris en charge nativement par .NET et pour stocker une instance de classe, il suffit d'utiliser la classe `JsonSerializer`.

Exemple de code de sérialisation d'un deck

```
try
{
    var json = JsonSerializer.Serialize(decks, new JsonSerializerOptions { WriteIndented = true });
    File.WriteAllText(decksFilePath, json);
}
catch (Exception ex)
{
    Console.WriteLine($"Error saving decks: {ex.Message}");
}
```

> *Deux éléments importants à relever*
>
> 1. *Il pourrait y avoir des erreurs => try/catch bienvenue*
> 2. *Pour faciliter la lecture par un humain, l'indentation est pratique*

Contraintes pour le json

Tant qu'on utilise des types standards, la sérialization est automatique. Ainsi, avec les éléments suivants :

```
private List<Deck> decks = new List<Deck>();
```

```
15 références
public class Deck
{
    9 références
    public string Id { get; set; }
    10 références
    public string Name { get; set; }
    17 références
    public List<Card> Cards { get; set; }
    1 référence
    public DateTime CreatedDate { get; set; }
    5 références
    public int TimesPlayed { get; set; }

    2 références
    public Deck()
    {
        Id = Guid.NewGuid().ToString();
        Name = string.Empty;
        Cards = new List<Card>();
        CreatedDate = DateTime.Now;
        TimesPlayed = 0;
    }
}
```

Lorsqu'on `sérialise` une liste de `Deck`, on obtient, par exemple, après avoir ajouté 2 cartes dans un paquet, le contenu suivant :

Contenu JSON


```
[
  {
    "Id": "e9e90b3c-05b5-49c3-b4aa-2fceeab85562",
    "Name": "ict-335",
    "Cards": [
      {
        "Id": "7723b6f1-9681-449e-8cfa-343d8eca9f39",
        "Recto": "1. Que veut dire l\u0027acronyme MAUI ?",
        "Verso": "Multiplatform Application User Interface",
        "TimesPlayed": 0,
        "TimesCorrect": 0,
        "TimesIncorrect": 0,
        "SuccessRate": 0
      },
      {
        "Id": "0097e449-64c7-4175-adb9-9e3d81822201",
        "Recto": "2. Comment cela se fait-il que le C# fonctionne sur Android ?",
        "Verso": "Un projet MAUI est transform\u00E9 en bytecode .NET et ex\u00E9cut\u00E9 sur un runtime .NET pr\u00E9vu pour Android",
        "TimesPlayed": 0,
        "TimesCorrect": 0,
        "TimesIncorrect": 0,
        "SuccessRate": 0
      }
    ],
    "CreatedDate": "2026-01-01T07:55:25.6055764+01:00",
    "TimesPlayed": 0
  }
]
```


Read

Une fois les données sauvegardées en `json`, on peut les charger de manière similaire avec le `JsonSerializer` :

```
private void LoadDecks()
{
    try
    {
        if (File.Exists(decksFilePath))
        {
            var json = File.ReadAllText(decksFilePath);
            decks = JsonSerializer.Deserialize<List<Deck>>(json) ?? new List<Deck>();
        }
        else
        {
            decks = new List<Deck>();
        }
    }
    catch
    {
        decks = new List<Deck>();
    }
}
```

> *La méthode `Deserialize` doit indiquer entre chevron (<>) le type de l'objet de destination. Dans notre cas, une liste de `Deck` a été sérialisée, c'est donc ce type là qu'on souhaite.*

> *De nouveau, plusieurs éléments indiquent que des erreurs peuvent survenir, dans l'exemple le choix a été fait de protéger le tout dans un bloc try/catch, puis de vérifier l'existence du fichier source (FileExist), puis de créer une liste vide si le fichier était vide.*

Update

Pour la mise à jour, nous avons déjà tout ce qui est nécessaire et il suffit de modifier un élément en mémoire puis de pérsister les éléments mis à jour :

```
private async void OnRenameDeckClicked(object sender, EventArgs e)
{
    if (currentDeck == null) return;

    string newName = await DisplayPromptAsync("Renommer", "Nouveau nom du paquet:",
        "Enregistrer", "Annuler", initialValue: currentDeck.Name);

    if (string.IsNullOrWhiteSpace(newName))
        return;

    currentDeck.Name = newName;
    dataService.UpdateDeck(currentDeck);
    DeckNameLabel.Text = newName;
}
```

> *L'instruction `dataService.UpdateDeck(currentDeck)` fera appel à `JsonSerializer.Serialize(...)` et ainsi le fichier JSON sera mis à jour avec le nouveau nom du deck...*

Delete

Là aussi, rien de particulier, il suffit de supprimer un élément de la liste, puis de réécrire le fichier JSON :

```
decks.RemoveAt(0); //retire le premier deck de la liste
var json = JsonSerializer.Serialize(decks, new JsonSerializerOptions { WriteIndented = true });
File.WriteAllText(decksFilePath, json);
```

Notes sur la suppression

Il est courant de ne pas réellement supprimer les données en informatique afin de pouvoir les restaurer...

Comment réaliser cela pour que les decks restent présents tout en étant masqués en cas de suppression ?

▶ Voir la réponse