Fiche de révision Génie Logiciel

En gros c'est une cheatsheet pour avoir les infos importantes sous la main

Fichiers et commandes

On commence par les fichiers et commandes à retenir pour le TP

Créer un projet de type API

```
dotnet tool install --global dotnet-ef --version 6.0.0
```

Installer EntityFramework

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0

dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 6.0.0
```

Fichier Context

```
using Microsoft.EntityFrameworkCore;
```

public class TodoContext : DbContext { public DbSet Todo { get; set; } = null!; public DbSet TodoLists { get; set; } = null!; public string DbPath { get; private set; }

```
public TodoContext()
{
    // Path to SQLite database file
    DbPath = "ApiTodo.db";
}

// The following configures EF to create a SQLite database file locally
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    // Use SQLite as database
    options.UseSqlite($"Data Source={DbPath}");
    // Optional: log SQL queries to console
    //options.LogTo(Console.WriteLine, new[] {
DbLoggerCategory.Database.Command.Name }, LogLevel.Information);
}
```

}

Exemple de Controller (modèle du prof)

using Microsoft.AspNetCore.Mvc; using Microsoft.EntityFrameworkCore;

namespace ApiTodo.Controllers;

[ApiController] [Route("[controller]")] public class TodoController : ControllerBase { private readonly TodoContext _context;

```
public TodoController(TodoContext context)
{
    _context = context;
}

[HttpGet]
public async Task<ActionResult<IEnumerable<Todo>>> GetItems()
{
    // Get items
    var items = _context.Todo;
    return await items.ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<Todo>> GetTodo(int id)
{
    var todo = await _context.Todo
        .Include(t => t.TodoList)
        .SingleOrDefaultAsync(t => t.Id == id);

    if (todo == null)
        return NotFound();

    return todo;
}

// POST: api/todo
[HttpPost]
public async Task<ActionResult<Todo>> PostTodo(Todo todo)
{
    _context.Todo.Add(todo);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetTodo), new { id = todo.Id }, todo);
}

[HttpPut("{id}")]
public async Task<IActionResult> PutTodo(int id, Todo todo)
```

```
{
    if (id != todo.Id)
    {
        return BadRequest();
    }

    _context.Entry(todo).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!_context.Todo.Any(e => e.Id == id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodo(int id)
{
    var todo = await _context.Todo.FindAsync(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.Todo.Remove(todo);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

}

Note importante sur les controllers

1. Si jamais un objet en **relation** est nul sur Swagger c'est que vous devez l'includes, voici un exemple d'includes : `var todo = await _context.Todo .Include(t => t.TodoList) .SingleOrDefaultAsync(t => t.Id == id);`
2. Pour le PUT, ne faites pas un New de l'objet, trouvez le puis modifier le. Sinon il y aura des erreurs.

Commandes pour les migrations

1. Pour créer la migration : `dotnet ef migrations add InitialCreate`
2. Pour appliquer la migration : `dotnet ef database update`
3. **SI JAMAIS** ça plante, supprimer le dossier migrations et le fichier .db puis refaites les deux étapes 😃.

Vocabulaire

SOLID

Le principe SOLID est un ensemble de cinq principes de conception de logiciels en programmation orientée objet, qui vise à rendre les logiciels plus compréhensibles, flexibles et maintenables. Ces principes ont été introduits par Robert C. Martin, souvent appelé "Uncle Bob". Voici une explication de chaque principe :

1. **S - Principe de Responsabilité Unique (Single Responsibility Principle)** : Ce principe stipule qu'une classe doit avoir une seule raison de changer. Cela signifie qu'une classe ne devrait avoir qu'une seule tâche ou responsabilité. En séparant les préoccupations, le code devient plus facile à maintenir et à comprendre.

2. **O - Principe Ouvert/Fermé (Open/Closed Principle)** : Les logiciels doivent être ouverts pour l'extension, mais fermés pour la modification. Cela signifie que vous devriez être en mesure d'ajouter de nouvelles fonctionnalités sans modifier le code existant. Cela est souvent réalisé en utilisant des interfaces ou des classes abstraites.

3. **L - Principe de Substitution de Liskov (Liskov Substitution Principle)** : Les objets d'une classe dérivée doivent être capables de remplacer des objets d'une classe de base sans affecter la correctitude du programme. Ce principe renforce l'héritage en s'assurant que les classes dérivées sont complètement substituables à leurs classes de base.

4. **I - Principe de Ségrégation des Interfaces (Interface Segregation Principle)** : Ce principe suggère qu'il vaut mieux avoir plusieurs interfaces spécifiques plutôt qu'une seule interface générale. Cela signifie qu'une classe ne devrait pas être forcée d'implémenter des interfaces et des méthodes qu'elle n'utilise pas.

5. **D - Principe d'Inversion de Dépendance (Dependency Inversion Principle)** : Ce principe recommande de dépendre d'abstractions, non de concrétisations. En d'autres termes, les modules de haut niveau ne devraient pas dépendre des modules de bas niveau, mais les deux devraient dépendre d'abstractions. Cela conduit à un couplage moins rigide et à un code plus modulaire.

Pourquoi JSON ?

L'utilisation de JSON (JavaScript Object Notation) est populaire pour plusieurs raisons, en particulier pour le stockage et l'échange de données. Voici quelques-unes des raisons principales :

1. **Format Léger** : JSON est un format léger pour le stockage et l'échange de données. Sa simplicité structurelle le rend facile à lire et à écrire, et il ne nécessite pas beaucoup d'espace, ce qui le rend efficace pour le transfert de données sur le réseau.

2. **Facile à Comprendre** : Le format JSON est facile à lire et à comprendre, non seulement pour les humains mais aussi pour les machines. Cela le rend idéal pour la transmission de données entre un serveur et une application web.

3. **Indépendant du Langage** : Bien que JSON soit dérivé de la syntaxe JavaScript, il est indépendant du langage de programmation. La plupart des langages de programmation modernes supportent JSON avec des bibliothèques intégrées ou externes, facilitant l'échange de données entre systèmes hétérogènes.

4. **Facilité d'analyse** : La plupart des environnements de développement offrent des outils d'analyse JSON intégrés ou facilement disponibles, permettant de transformer les données JSON en objets utilisables dans le code de programmation.

5. **Structure Flexible** : JSON peut représenter des structures de données complexes, y compris des objets et des tableaux. Cette flexibilité permet de modéliser un large éventail de données.

6. **Support Web Natif** : JSON est extrêmement populaire dans le développement web. Étant donné que le format est nativement compris par les navigateurs web et JavaScript, il est largement utilisé pour l'échange de données dans les applications web.

7. **Performance** : JSON est souvent plus rapide à analyser et à manipuler par rapport à d'autres formats de données comme XML. Ceci est crucial dans les applications web où la vitesse et l'efficacité sont essentielles.

Diagramme de séquence

Un diagramme de séquence en UML (Unified Modeling Language) est un type de diagramme qui est utilisé pour illustrer comment les objets interagissent dans une application au fil du temps. Il met l'accent sur la séquence temporelle des messages entre les objets.

Voici les éléments clés et les caractéristiques d'un diagramme de séquence :

1. **Objets et Acteurs** : Les objets (instances de classes) et les acteurs (entités externes, souvent des utilisateurs) sont représentés par des rectangles en haut du diagramme. Chaque objet ou acteur a une ligne de vie qui descend verticalement.

2. **Ligne de Vie** : La ligne de vie, représentée par une ligne verticale pointillée sous chaque objet ou acteur, illustre la présence de cet objet pendant la période de temps couverte par le diagramme.

3. **Messages** : Les interactions entre les objets sont représentées par des flèches horizontales, appelées messages. Ces flèches montrent le flux de messages (appels de méthodes, envoi de signaux, etc.) d'un objet à l'autre. L'ordre des messages est représenté de haut en bas.

4. **Activation** : Un rectangle fin le long de la ligne de vie, appelé "activation", montre la période pendant laquelle un objet exécute une opération. La taille de l'activation peut indiquer la durée d'exécution de cette opération.

5. **Synchronisation et Asynchronisme** : Les diagrammes de séquence peuvent représenter des appels synchrones (où l'expéditeur attend une réponse avant de continuer) et des appels asynchrones (où l'expéditeur continue sans attendre).

6. **Décisions et Branchement** : Bien que moins fréquemment utilisés dans les diagrammes de séquence, les conditions et les choix peuvent être représentés, montrant comment les différentes séquences d'actions peuvent se produire en fonction de certaines conditions.

7. **Focus sur le Temps** : Contrairement à d'autres types de diagrammes UML, les diagrammes de séquence se concentrent sur le temps. Les événements sont ordonnés chronologiquement de haut en bas.

Diagramme d'activité

Le diagramme d'activité en UML (Unified Modeling Language) est utilisé pour modéliser le flux de contrôle ou le flux de travail entre divers éléments d'un système. Il est semblable à un diagramme de flux traditionnel mais offre des capacités plus avancées et spécifiques pour représenter le comportement d'un système. Voici les caractéristiques principales d'un diagramme d'activité :

1. **Activités** : Les activités sont des actions réalisées par le système. Elles sont représentées par des rectangles arrondis et indiquent les tâches ou les opérations effectuées.

2. **Transitions** : Les flèches représentent les transitions d'une activité à l'autre. Elles indiquent le flux de contrôle ou l'ordre des opérations.

3. **Nœuds de Décision et de Fusion** : Les nœuds de décision (représentés par des diamants) sont utilisés pour montrer les points de bifurcation où des décisions sont prises pour choisir le chemin à suivre. Les nœuds de fusion rassemblent les chemins divergents.

4. **Nœuds de Début et de Fin** : Chaque diagramme d'activité a un nœud de début (un petit cercle plein) et un ou plusieurs nœuds de fin (un cercle avec un cercle plus petit à l'intérieur).

5. **Parallélisme (Fork et Join)** : Les barres horizontales ou verticales ("Fork" pour diviser et "Join" pour fusionner) représentent l'exécution parallèle de plusieurs activités. Elles permettent de modéliser les situations où des processus se déroulent en parallèle.

6. **Piscines et Lignes de Natation (Swimlanes)** : Les "swimlanes" sont des zones qui regroupent les activités par acteur ou par rôle (comme des départements dans une entreprise). Ils aident à organiser les activités en fonction de qui les exécute ou de qui est responsable.

7. **Artéfacts** : D'autres éléments comme les objets, les données ou les artefacts peuvent être inclus pour montrer comment ils sont manipulés ou affectés par les activités.

8. **Points de Synchronisation** : Les diagrammes d'activité peuvent inclure des points de synchronisation où plusieurs flux convergent et attendent que tous les flux atteignent ce point avant de continuer.

Les diagrammes d'activité sont largement utilisés pour modéliser les processus d'affaires, les flux de travail des logiciels, et pour détailler les algorithmes complexes. Ils sont particulièrement utiles dans la modélisation de scénarios où la séquence d'actions et les décisions conditionnelles sont essentielles.