Compte Rendu SAE 2.01 ShopNCook

par TUAILLON leo et BATISTA Maxime

Le module `Models` (aucune dépendance), diagramme ICI
Ce module contient les définitions des modèles qui sont les entités centrales de l'application. Comme Account, User, Recipe, etc

Le module `Services` (dépendances: Models), diagramme ICI
Ce module contient toutes les abstractions qui forment les services. Il y a plusieurs types de services. Un service pour l'authentification des utilisateurs, un service pour gérer les recettes globales, il y a aussi des services qui sont reliés à un compte, comme un service pour gérer les recettes uploadés par un utilisateur, ou un service qui gère ses préférences et les recettes qui pourraient lui être recommandés.
Ce module est la pièce centrale de la logique fonctionnelle du projet, son implémentation est chargée de vérifier la validité des données insérées, et des actions que le module `ShopNCook` tente d'effectuer.

Le module `ShopNCook` (dépendances: MAUI, MAUI Toolkit, Models, Services), diagramme global ici
C'est le seul module du projet qui dépend du framework MAUI car c'est le seul à être en contact avec l'utilisateur. Il contient les vues graphiques et son rôle est d'utiliser les services mis à disposition par le module `Services` et de les adapter graphiquement pour l'utilisateur

Le module `LocalServices` (dépendances: Models, Services), diagramme ici
Ce module est une implémentation du module `Services` et contient une implémentation simple, où les données sont stockées directement sur l'ordinateur en JSON.

Immuabilité et encapsulation

Nous avons opté pour rendre les modèles immuables ainsi que les structures de données retournées par les `Services` : (`ImmutableDictionnary`, `ImmutableList` etc).
Les services sont mutables mais toutes les données qu'ils retournent et que l'on leurs donne sont immuables. Pour effectuer une action mutable, il faut donc obligatoirement passer par le service.
l'immuabilité rend le débogage plus simple, car les objets manipulés sont plus prédictible, et cela renforce encore plus l'encapsulation au niveau des services :
Pour donner un contrôle absolu au Services sur les données manipulées il est préférable d'obliger de les utiliser directement pour chaque modification, par exemple, si un utilisateur publie une recette, il faudra passer par le service `IAccountOwnedRecipesService` qui gère les recettes crées par l'utilisateur, et par la méthode `UploadRecipe` de celle-ci, ainsi, l'implémentation a le contrôle sur quelles sont les recettes qui sont valides à l'upload (on pourrait imaginer une implémentation qui refuse une recette si les étapes contiennent des

insultes, ou si l'image d'illustration n'a pas le bon format etc. Ce n'est pas le cas de l'implémentation actuelle).

Gestion des erreures

Definition / emission

La manière dont sont retournées les erreurs fonctionnelles est définie par le Module `Services`. La définition actuelle est très simple, si le service refuse l'action, il retourne null s'il fallait retourner un objet (ex: le login à un compte, qui retourne un objet Account si la connection est effectuée, n'a pas le bon mot de passe), ou un booléen `false`.
Cette manière a l'avantage d'être simple mais ne permet pas au module ShopNCook d'obtenir du contexte sur l'erreure ce qui le contraint à rester global lorsqu'il adapte l'erreure pour l'afficher à l'utilisateur.
Nous aurions pu imaginer définir des exceptions pour chaque type d'erreurs, ce qui aurait donné plus de contexte à l'application pour afficher des erreurs plus poussées à l'utilisateur. Malheureusement, nous manquions de temps pour aboutir à cette amélioration.

Interception / Affichage

Les erreurs sont interceptées par le module ShopNCook qui se charge de l'adapter pour afficher à l'utilisateur les éléments qui causent le problème.
Actuellement, cet affichage est très simple puisque les erreures sont affichés en utilisant un Toast, statiquement encapsulé par la classe `UserNotifier`, qui contient deux niveau d'erreures: Warn, pour les erreures non fatales (ex: les identifiant sont invalides), et Error pour les erreures fatales (ex: une recette n'a pas pu être supprimée car elle n'existe pas)

Persistence des données

[voir le diagramme UML ici]
[voir le diagramme de paquetage ici]
La persistance des données est gérée par l'implémentation des services. Dans l'implémentation `LocalServices`, c'est le namespace `LocalServices.Data` qui contient les classes chargés de la persistance. Nous avons interfacé la base de données via l'interface `IDatabase` pour pouvoir effectuer des mocks lors des tests unitaires, mais en pratique c'est l'implémentation `CatastrophicPerformancesDatabase` qui est utilisée.
Cette implémentation persiste les données dans des fichiers json, qui sont réécrits à chaque modification de la base de données, d'où son nom.

Patrons de conception

Builder

Le modèle `Recipe` étant un objet relativement complexe, un builder a été fait pour faciliter sa création (voir `RecipeBuilder`)

Diagrammes de séquence

Ici, le diagramme de séquence pour la connection de l'utilisateur à un compte
https://codefirst.iut.uca.fr/git/ShopNCook/ShopNCook/src/branch/master/documentation/diagrammes/Sequence%20login.svg

Ici, le diagramme de séquence pour la recherche d'une recette ou plusieurs recettes
https://codefirst.iut.uca.fr/git/ShopNCook/ShopNCook/src/branch/master/documentation/diagrammes/Sequence%20recipe%20search.svg