



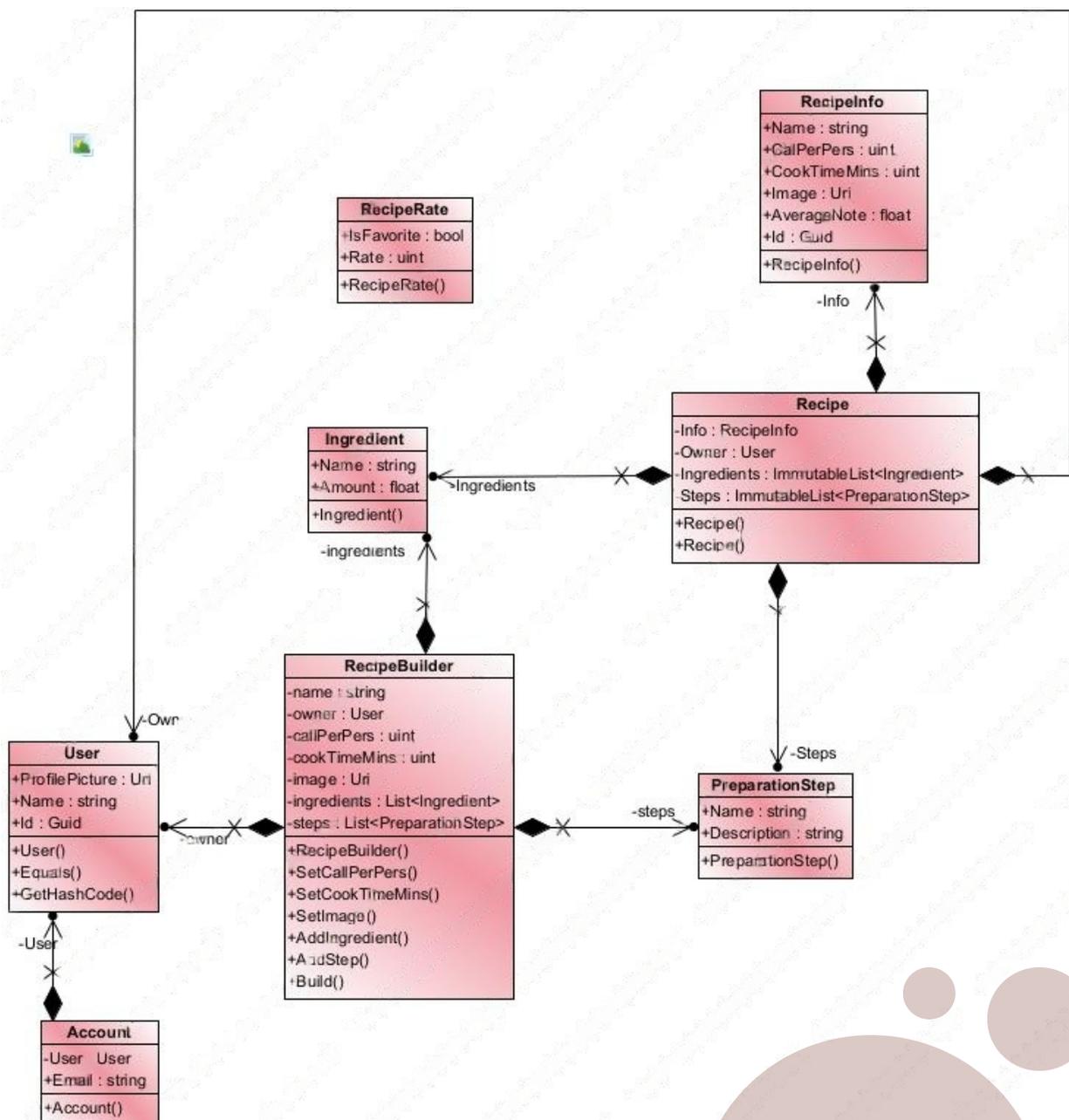
# **Compte rendu ShopNCook :**

**PAR TUAILLON LEO ET BATISTA MAXIME**



## Le module `Models` (aucune dépendance) :

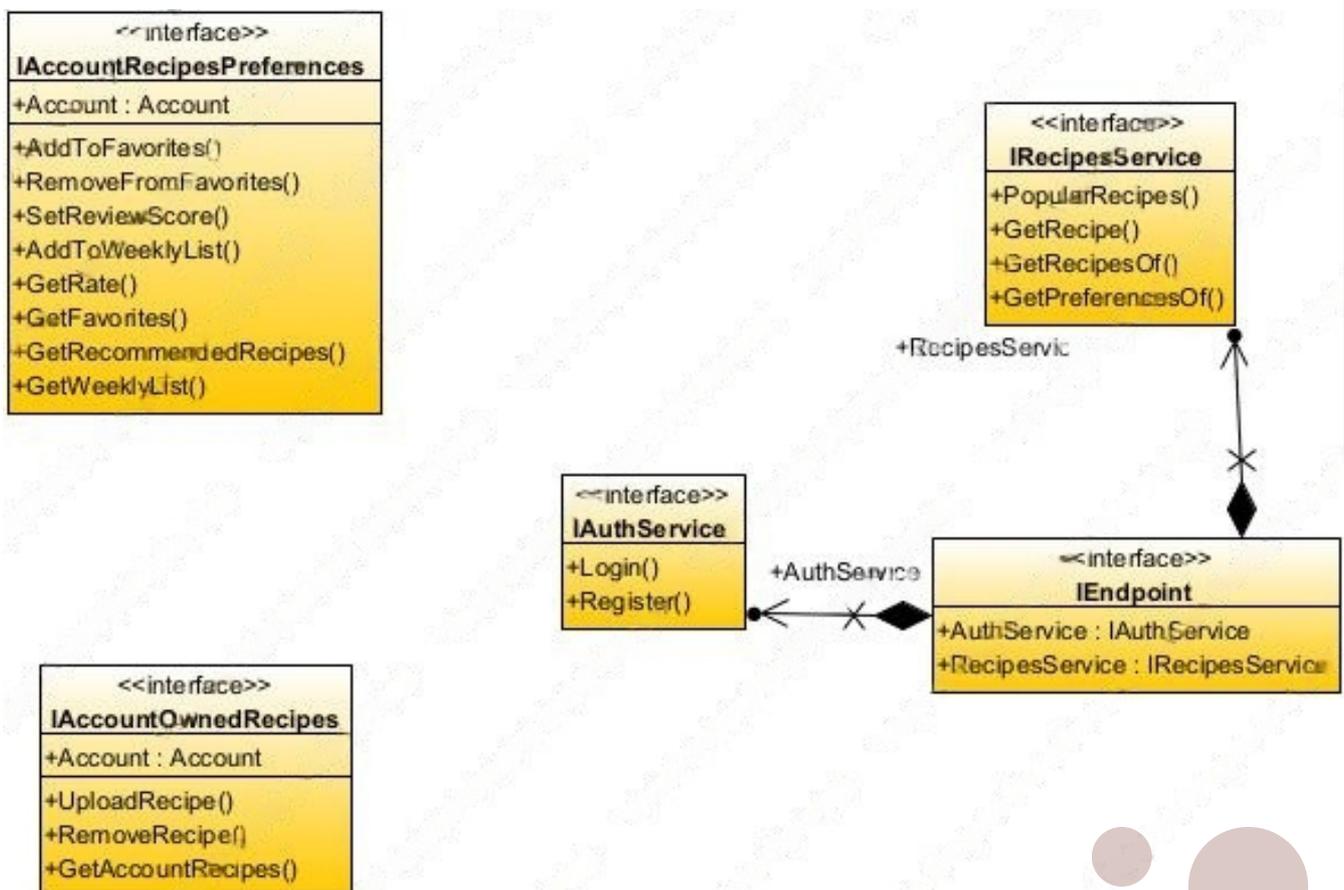
Ce module contient les définitions des modèles qui sont les entités centrales de l'application. Comme Account, User, Recipe, etc...





## Le module `Services` (dépendances: Models) :

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.



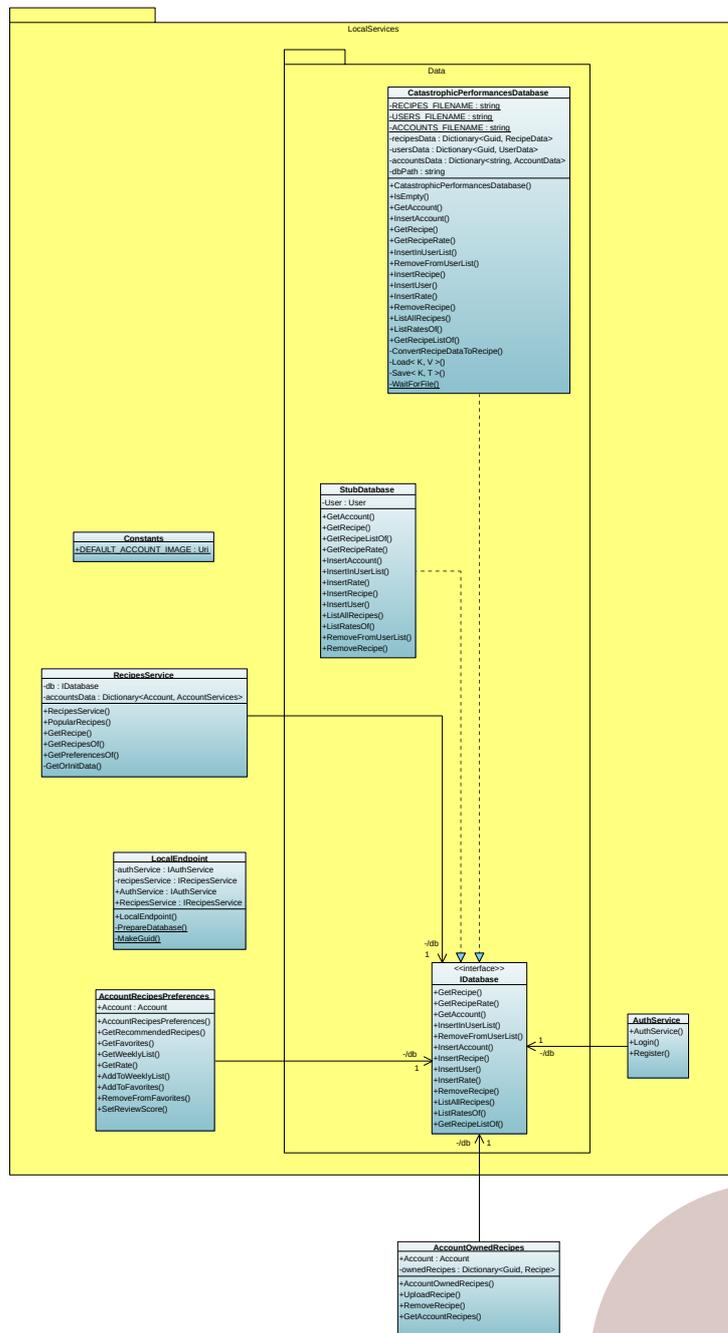




## Le module `LocalServices` (dépendances : Models, Services) :

Ce module est une implémentation du module `Services` et contient une implémentation simple, où les données sont stockées directement sur l'ordinateur en JSON.

Lien vers le diagramme en plus grand : [Cliquez ici!](#)





## **Immuabilité et encapsulation :**

Nous avons opté pour rendre les modèles immuables ainsi que les structures de données retournées par les `Services` :(` Immutable Dictionary`, `ImmutableList` etc). Les services sont mutables mais toutes les données qu'ils retournent et que l'on leur 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 créés 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 erreurs" : Définition / Émission :

### Definition/emission :

La manière dont les erreurs fonctionnelles sont retournées est définie par le module **Services**. La définition actuelle est assez simple : si le service refuse l'action, il retourne null s'il devait retourner un objet (par exemple, le login à un compte, qui retourne un objet Account si la connexion est effectuée, n'a pas le bon mot de passe), ou un booléen **false**. Cette méthode a l'avantage d'être simple, mais elle ne permet pas au module Shop N Cook d'obtenir du contexte sur l'erreur, ce qui le contraint à rester global lorsqu'il adapte l'erreur pour l'afficher à l'utilisateur.

Nous aurions pu envisager de définir des exceptions pour chaque type d'erreur, ce qui aurait donné plus de contexte à l'application pour afficher des erreurs plus détaillées à l'utilisateur. Malheureusement, nous manquons de temps pour aboutir à cette amélioration.

### Interception / Affichage :

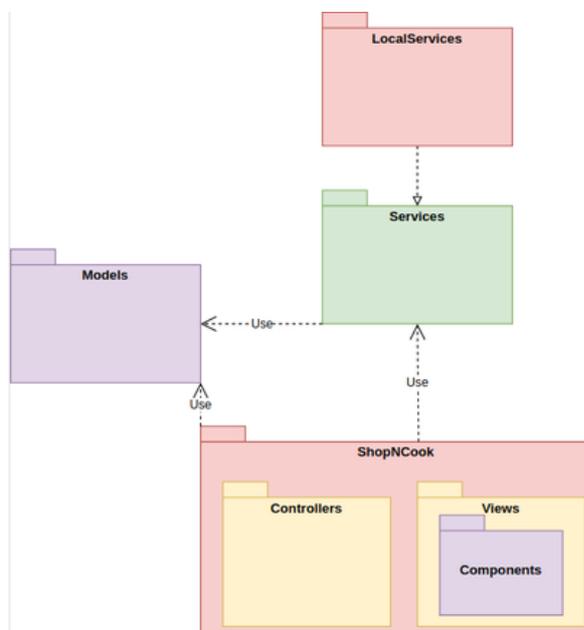
Les erreurs sont interceptées par le module Shop N Cook qui se charge de les adapter pour afficher à l'utilisateur les éléments qui causent le problème. Actuellement, cet affichage est très simple puisque les erreurs sont affichées en utilisant un Toast, statiquement encapsulé par la classe **UserNotifier**, qui contient deux niveaux d'erreurs : **Warn**, pour les erreurs non fatales (par exemple, les identifiants sont invalides), et **Error** pour les erreurs fatales (par exemple, une recette n'a pas pu être supprimée car elle n'existe pas)."



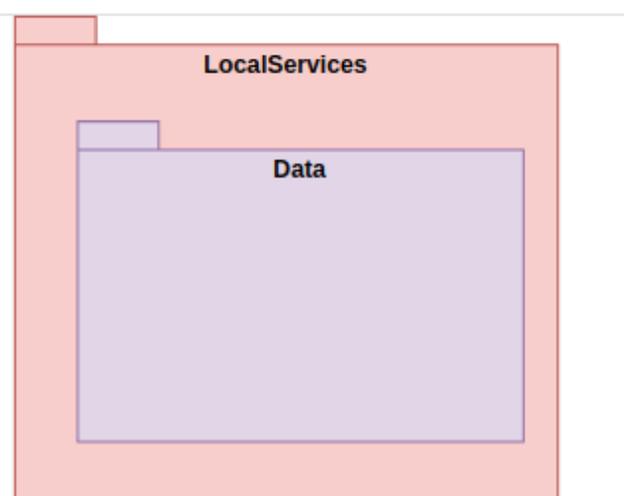
## Persistence des données :

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 chargées 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.

## Diagramme de paquetage :



## Diagramme persistance :





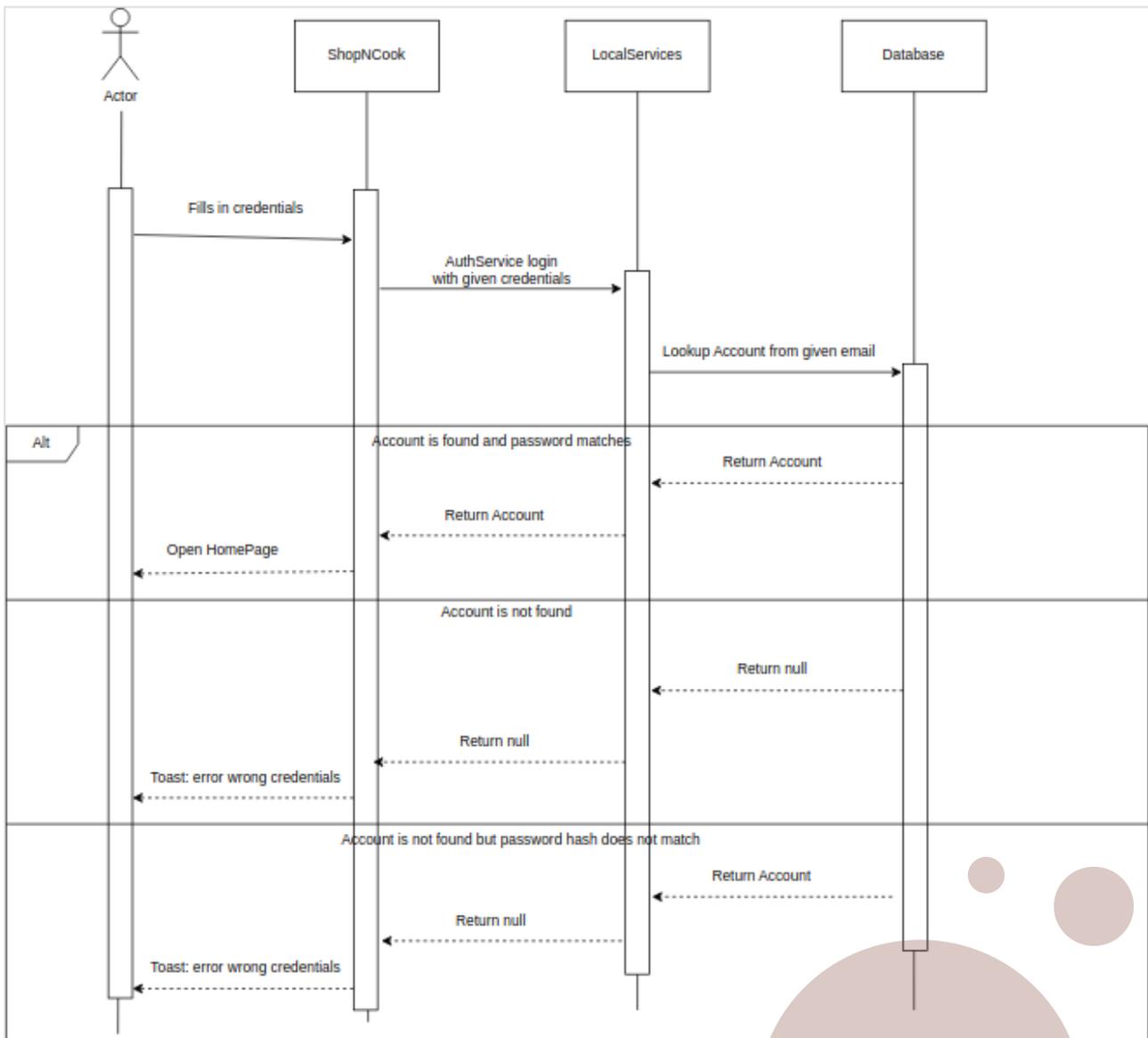
## Patrons de conception

### Builder

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

### Diagrammes de séquence :

Voici le diagramme de séquence pour la connexion de l'utilisateur à un compte.





## Diagrammes de séquence :

Voici le diagramme de séquence pour la recherche d'une recette ou plusieurs recettes.

