FoodRecipe – Documentation

"FoodRecipe" is a web application that allows users to search for various recipes to make food. In an effort to ease searching the application allows the user to specify filters in order to reduce the total amount of recipes from which to pick.

- Recipe type (e.g.: dessert, main course, soups and so on)
- Approximate time for preparation.
- Required ingredients.
    - Optionally their quantity.
- Excluded ingredients (allergies or they simply do not like how they taste).

Since all filtering can be resumed to answering clear questions with yes or no, or in other words true or false, and the fact that data is never changed a declarative approach seems best. Declarative techniques strive against change and promote immutability, but the core focus is into specifying what needs to be done to reach a goal and skip all the details of how it is done emphasizing functional requirements.

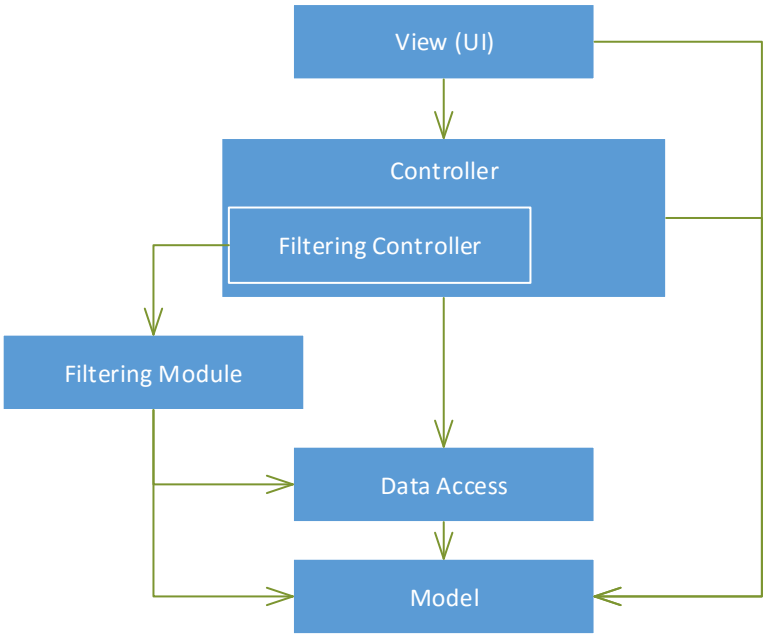
Filtering is specifying a set of parameters that are going to be provided to a number of rules in addition to the items that are being filtered. If a given item satisfies the rules then it gets past the filter, simple as that. It all resumes to true or false which allows the use of logical programming techniques. The greatest benefit of it is that for the part which specifies the rules can be written in a *domain specific language* from which code can be generated and used within the application. A *DSL*[1] is designed to solve a specific class of problems and does not try to address multiple domains making it easy to learn and use not to mention that generally there more limitations there are the easier it is to optimize.

A part of the application will be written using C# and ASP.NET MVC and Azure Table Storage. For the filtering part it will use Prolog, more specifically P#. This is an implementation for Prolog that makes use of Microsoft's .NET framework, one can write predicates that use types defined in C# and then use those predicates further more in C# or any other *CLS*[2] compliant language for that matter (e.g.: Visual Basic .NET, F#, IronPython and so on).

P# comes with a tool that allows predicates to be written and executed within an interactive application but as well as generating C# code for all defined predicates. The engine that runs those predicates is written in .NET meaning it can be used from C#. In the end the application is fully written in C# only that the code corresponding to predicates are generated. This helps with debugging and overall development.

On the next page there is a diagram illustrating the overall architectural approach.

[1] DSL – Domain Specific Language.
[2] CLS – Common Language Specification.



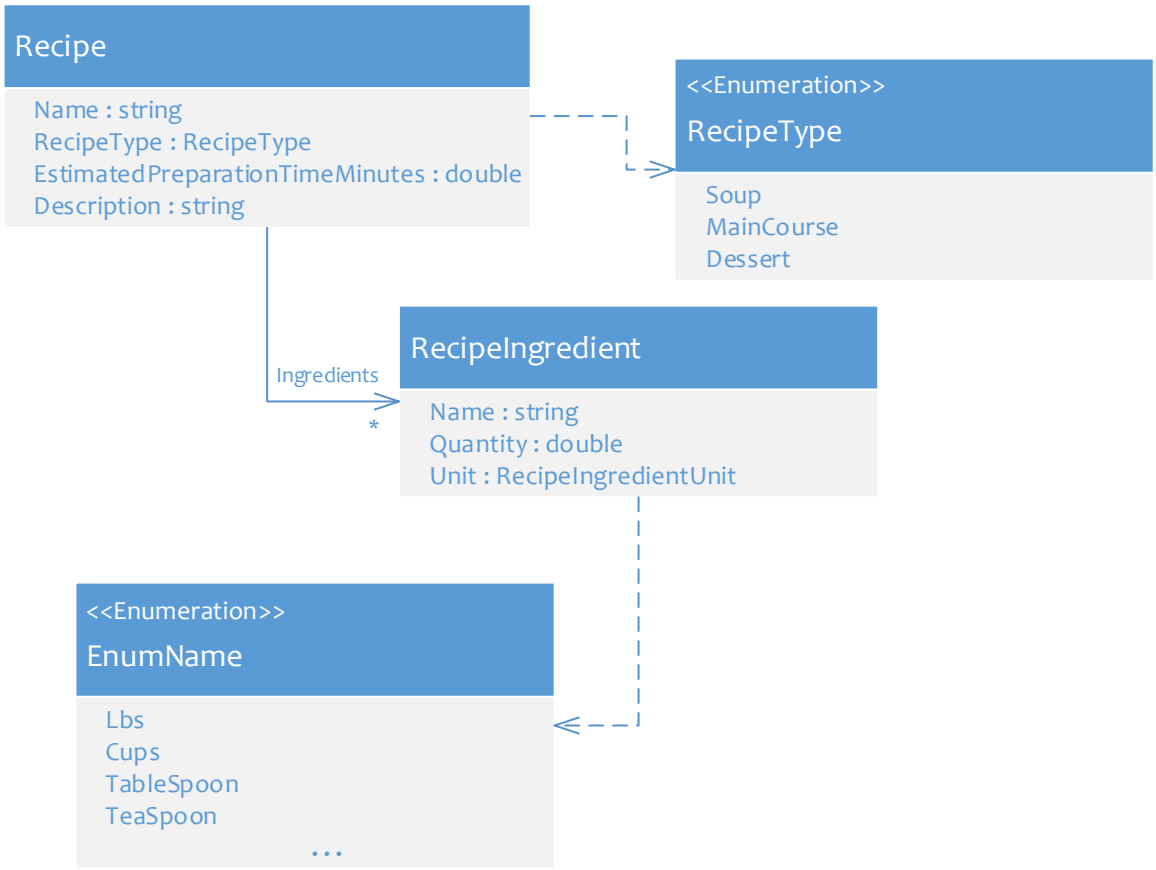
The model is the central part, everything is built around it. The *Data Access* are a set of interfaces for retrieving model objects from a storage system. This application will be using Azure Table Storage as that is cheaper than an SQL database and the little abstractions done at this moment will allow to easily migrate from one storage system to another.

The *Filtering Module* has a few abstractions of its own that make it look like a repository. It uses *Data Access* to retrieve *Recipes* but it also performs some operations with them. The approach is to use predicates that are defined on model objects to retrieve only what is asked for.

At the *Controller* level there will be dedicated one that intermediates between the user and his/hers input and the *Filtering Module*.

The *View*, like all other layers makes use of the model in order to display it. This is strictly for how information is shown and not how it is filtered, stored or retrieved.

Below is a UML[1] class diagram illustrating the business model.

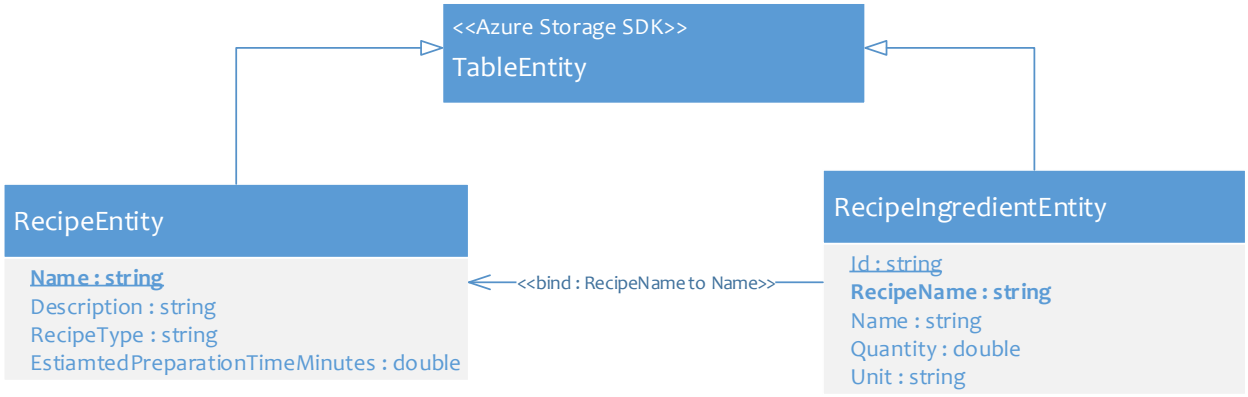


The data is stored using *Azure Table Storage*, for this a set of *Data Transfer Objects* had to be created in order to avoid mudding the business model. In short the association is inverted, from *RecipeIngredient* to *Recipe* and all enumeration type members have been replaced with their string representations (to ease debugging, it is easier to see that *'Soup'* was stored instead of *integer zero* which who knows what it means).

Besides this, *Azure Table Storage* has, for each table, a row key and at least one partition key. A partition key + row key provides a unique identifier for a row in the table. In other words, partition keys group rows into partitions and the row key identifies a single row in a partition. This is useful when dealing with aggregated entities (e.g. *RecipientIngredient*) as all aggregated entities can share the same partition key grouping them together and an auto generated row key so each can be queried uniquely if necessary (usually in the case of update or delete operations). The partition key is used by *Azure Table Storage* to facilitate fast access by avoiding a table scan, this is why it is useful.

On the next page a UML class diagram illustrates the *Data Access Objects* and the associations between them. Partition keys are bolded while row keys are underlined. Only one partition key was used per table, each class maps to a table in *Azure Table Storage*.

[1] UML – Unified Modeling Language.



Note that *RecipeEntity* has the row key equal to the partition key at all times. This makes the name of the recipe unique throughout the table, exactly how a unique key or primary key in SQL would. On the other hand, *RecipeIngredientEntity* has *Id* as row key and *RecipeName* as partition key. This groups ingredients by recipe meaning that when a recipe is required to be loaded the ingredients can be quickly retrieved by using just the recipe name making loading of recipes optimal with this storage.

Filtering is done exclusively through P#. The language itself and the C# code generator is a bit dodgy but does the job quite well even though it lacks a detailed documentation or some features for iterating over collections. To ease development a custom tool was written that takes P# source files and generates a C# source file containing the translated Prolog predicates. The custom tool does nothing more than invoke the available tool for generating C# source files from predicates and concatenates all files into one. Whenever the file containing the predicates is saved the equivalent C# source code is generated.

The filtering module is quite simple. It uses a repository that retrieves Recipes and P# generated predicates. There is a predicate that tells whether a Recipe fulfils a filtering criteria. If it does, then that Recipe is added to the result collection. Below is a UML diagram illustrating the filtering object models.