On how to implement the database schema

Since we are working with a NoSQL DBMS, to enforce relations between different entities is a little different. To retain the referential integrity, the solution may add extra cost. Considering the nature of this project, I choose to relax the data consistency a little bit — use non-atomic operation to cleanup/insert parent/child record. The application logic will have to be aware of maintaining referential relationship between related tables. This way the back-end doesn't need to check the integrity of the record in each query hence avoid the extra cost. This may allow the possibility that a user remains incorrectly active after its removal within 2 atomic operations. Luckily, this situation is extremely unlikely to happen in this project. The concept of data transaction is not that strict here.

In the big picture, I separate the software into three layers, the same as the standard MVC pattern. In the Model layer, I create one manager class for each entity (table). To enforce the relationship between each entity, I build a wrapper manager class on top of a set of related entities. For example, f

- The **AccountManager** class is responsible for handling **Profile**, **AdminRecord**, **Provider** and **Patient**. It's sensible to do so as they are all 1 to 1 relations with total participation constraint around **AdminRecord**.
- The creation of **HistoryAction** and **Identity** are handled separately because they are in a 1 to N relationship with **AdminRecord** without requiring to be in the relationship, but their removal is dealt in the **AccountManager** as they are deleted in a cascaded manner.
- The **SessionManager** handles the relation between **Provider** and **Patient**. Because they are in a M to N relationship, an extra table is required naturally to represent the relationship even in SQL. However, we happen to have a **Session** entity that can by-the-way achieve such job very nicely.

The wrapper manager performs tasks to insert/remove records to/from each of its derivative tables to maintain referential integrity. In the Controller layer, I force all the APIs, either public or private, to use only the wrapper Manager to ensure that the relation cannot be invalidated by foreign agents. The front-end call the controller class indirectly via the Meteor methods call. The Controller class does not interact with each other in order to eliminate dependencies on business logic layer. Only the wrapper Managers have the knowledge of how each entity is related among each other.

NoSQL, indeed, is not a good solution for relational model. However, at the time of my development, it is the only choice in the Meteor framework. Having chosen Meteor might be more or less a mistake, I shouldn't have used immature product in actual production. It's fortunate that this only causes minor issue. Even so, NoSQL DBMS still provides an excellent shortcut to represent hierarchies. In representing different measurements, I am able to crunch all those remotely different measurement entities into one big table with NoSQL DB. This saves quite some extra steps to manage those different types of data. It's a win here.

Chifeng Wen — May 1st 2016