- Author: Petre Iordanescu
- Publisher: RENware Systems (www.renware.eu (http://www.renware.eu)), research & innovation department
- Edition: 2021 August 21
- Language: EN

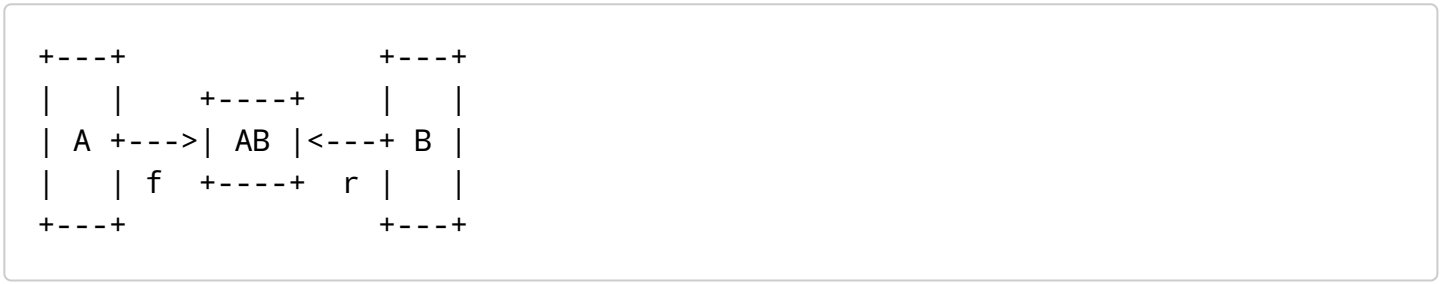
About brief series. These materials are derived from author training sessions as course notes for students, with the intention for presentation of basic topics regarding a concept. These materials present a concept, what is good for, its main topics and what to deep research to find out and learn more about it.

Relational to JSON. Many to many related entities

If it is the first contact with relational model in connection with JSON model, is recommended to read before reference 0.

The relational model

In relational model, a many to many (will use *m2m* or *m:m* notation, both with the same meaning) relationships are implemented as an intersection entity between entities that need to be in *m2m* relationship, as shown in the following diagram.



Notations on diagram represents:

- **A** and **B** are the entities that need to be in *m2m* relationship
- **AB** is a helper entity (named **intersection**), needed to represent *m2m* relationship. Otherwise, as it is in diagram, it has no 'real life' representation
- **f** and **r** are the business names of relationships, *f* from A to B and *r* from B to A (from forward and reverse)
- an *arrow* shows the 'many' termination of a relationship

Remarks

- in all real life scenarios, a *m2m* relationship makes sense just when is read in *f* direction or in *r* direction.
- as shown in diagram, the intersection is a 'pure' one, which can br found in very abstract models and contains just the primary keyd from A and B
- in 'real life' the intersections contains also some ***useful values*** (see example below)

A real example

Let's take as example a *m2m* relationship between some *payable invoices* (as **A**) and some *payments* (as **B**) makes sense when is read:

- in *f* direction: an invoice is paid by some payments
- in *r* direction: a payment is done for some invoices

In this example, the **AB** intersection will contain a useful (not only the primary keys of **A** and **B**) attribute. Will contain the ***value paid*** by a payment for one invoice (that related).

JSON models, Generalities

The JSON model is more flexible and you can represent tne relational *m2m* relationship in more ways. Please do ot forget that JSON model cannot by default (itself) assure the integrity of relationships as in many relational database servers you can do (without problems).

So, the representation in JSON could be (only the usual formats are enumerated here):

- **T1** - purely as complete imitation of relational model
- **T2** - follow *f* and *r* directions

T1 model

T1 model is probably the most "complex" one but it have the advantages of SQL normalised structures, more exactly non duplication of data. Basically (using the same object names as described in relational model) the things looks like that:

```
{
  "A": [
    {
       "pk_A": <val>,
       "other columns": ...
    }, ...other A recs...
  ],
  "B": [
    {
       "pk_B": <val>,
       "other columns": ...
    }, ...other B recs...
  ],
  "AB": [
    {
       "pk_A": <val>,
       "pk_B": <val>,
       "value_paid": <val>
    }, ...other AB recs...
  ]
}
```

T1 explained

- the JSON is a big object (to be compacted as 1 object) containing 3 distinct objects corresponding to the 3 tables from relational model, A, B and AB.
- every A, B and AB object contains a collection of records, here implemented as lists, but this is NOT a MUST (the representation as list)
- in every collection of records corresponding to each object-table, the order in list should not be relevant and should not be used in any way
- records collection contains an object with columns of the record and their values. The key is inside object (dictionary) as the other columns, but if it is single (ie, not compounded as in AB) cukd be put outside the object and separated of the other columns. This will break the rule for JSON to be an exactly imitation of relational but is perfectly feasible
- being a dictionary (set of columns), the order of keys is not relevant and in fact is not guaranteed by dictionary definition
- finally, must be said that are clearly feasible more variations and deviations from shown structure, but the main objective as BEING A PERFECT IMITATION of relational probably will not be respected in full. As I already said, this should be a must only for integration with l other systems and for representation of data more formal and standard

Remarks on T1 model

- querying T1 model is a little bit difficult, but not hard enough to be used frequently in real implementations
- being si closed to relational model is easy to implement corresponding DML
- need much stuff (at least as much as relational) to show some relevant information in UI

T2 model

The T2 model is more appropriate to programming style and therefore is more denormalized from the relational point of view.

There are more possibilities to represent a *m2m* relation as *T2* (because the freedom of JSON, dictionaries and arrays flexibility). Here we chosen a model that follows **directions of information**, respectively *f* and *r*.

Also, must be noted that we followed both directions, bur in practise could be followed just one, depending on application / system goals. Also, must be said that if the designed system is a general API for example, both views should be presented ad you cannot know what "intention" have the consumer of this information.

```
{
  "A": [
    {
       "pk_A": <val>,
       "other columns": ...
       "f_data": [<ref1>, ...],
    }, ...other A recs...
  ],
 "B": [
    {
       "pk_B": <val>,
       "other columns": ...
       "r_data": [<ref2>, ...],
    } ...other N recs...
}
```

T2 model explained

- the T2 model contains in JSON only the base entities, **A** and **B**
- for every record of A or B, there are **f_data** and **r_data**, named so from *forward* and *reverse* relationships just to be like in relational used example. Otherwise, you can name them with something relevant in your work, and we encourage this as reflecting the right use cases you need to implement
- the `<ref1>` and `<ref2>` lists contains directly the (scalar) values related through **AB** intersection and if is relevant or at least useful in your work you can use for list values not a scalar value, but a dictionary with primary key from the other table too
- as example for `<ref1>`, in the complex form, could be:

```
[
  {
    "pk_B": <val1>,
    "data_B": <val2>,
    "data_AB": <val3>
  }, ...other recs from AB...
]
```

- **`val1`** is the pk from related entity
- **`val2`** is yhe value from related entity
- **`val3`** is the value from intersection entity, if there is one (other than keys)

- as example for **`<ref1>`**, in the simplest form, could be:

```
[ <val_2> or <val_3>, ... ]
```

- with values having the same meaning
- is recomanded to not mix different values, from A with those from AB (is an anti-pattern or a misuse) the values ***NOT BEING NAMED***. Instead of that, use a more complex form like in previous example

Remarks on T2 model

- this model is easier to use as end-user relevant data source in UI
- data from this model can be "harder" to sync back in relational database or even impossible (as consistent) in the JSON simplest form
- clearly choose the version that covers requirements, but always "keep 1 move ahead" and think for extendability and for maintainability

---

Notes and biographical references

- 0. Relational to JSON representation article (https://learning.renware.eu)