This post outlines a way of working with data in JavaScript. I received a task of developing a Calendar app, cells of which would actually become editable forms. Moreover, users would get to choose which specific form to use for each cell. Because of that, a cell pattern had to be generated on the client-side according to this choice. The intricacy here was that fields of each form ought to be editable as well. So, clearly, all changes needed to be stored somewhere.

I abandoned the concept of creating form patterns, because upon modification of one field another had to change, too. Field interaction logics concerned three database tables:

- Two of them at least half-filled with data;
- One for storing changes.

That's why I decided to create my own storage (as a **Store** in **Ext JS**). The main idea behind it is that each **JSON** format node is presented as a separate unit. The principle is: I get all three tables and create three index trees, where each parent is also a context of a subsidiary node. Perhaps, it sounds a tad complicated, take a look at this example:

**Customers**

```
var                    Customers                    =                    [
  {
```



+44 20 7183 5820

info@magora.co.uk
sales@magora.co.uk

```
    },
```

**Orders**

```
var orders = [
  {
```

**Schedule**

```
var schedule = [
  {
```



+44 20 7183 5820

info@magora.co.uk
sales@magora.co.uk

```
```

There is a catalogue of customers, a list of orders and a shipping schedule. This schedule can be edited in the Calendar and Couriers can see orders assigned to them there. A courier can also be added to/removed from the schedule on a weekly basis.

## Implementation

**Store** object is responsible for storing data and transferring it to/from the server. Implementation proceeds as follows:

```
var Store = function(){
```

**InMemory** function allows adding an object or an array received from the server to an index tree:

```
function inMemory(from)
```


+44 20 7183 5820
info@magora.co.uk
sales@magora.co.uk

```
                    }
                }
            })(_innerValue);
```



+44 20 7183 5820

info@magora.co.uk
sales@magora.co.uk

```
```

Using **get**, **set** or any other node function, we gain access to each level. For instance, you can implement the **first** method for the **facadeArray**. Then getting a specific courier's schedule for a certain week will look like this:

```
var firstOnWeekPredicate = function(x){
    return x.get().CustomerId.get() == "1" && x.get().StartWeek.get() ==
```

If you haven't yet worked with such handy functions as **where**, **first**, **last**, **union**, **map**, etc. - check out underscorejs.org.
Now you can modify **scheduleByCourierIdAndWeek** array: add a new day of the week or change existing ones.

```
scheduleBy?ourierIdAndWeek.push({
```

The "facade" of **getNode** function can be reworked according to your needs.

## Result

So, what is the end result here? We've divided a big object, received from the server, into connected nodes. We've sent nodes to the segments of code responsible for their processing. We've converted data the way user desired. Now we need to put it all together into a single object and send it back to the server. For that we utilize **dumpAcc** function:

```
function dumpAcc(storeAcc) {
```


+44 20 7183 5820
info@magora.co.uk
sales@magora.co.uk

```
            }
```

**JSON.stringify(dumpAcc(store.getScheduleList()))** allows us to get **clientResponse**.
And, lastly, here are functions **clone** and **isPrimitive** that I've also used.

```
function                                clone(a)                                {
```

```
        return typeof (value) === "string" || typeof (value) === "number"
```



+44 20 7183 5820

info@magora.co.uk
sales@magora.co.uk