

# CLASSMAKER

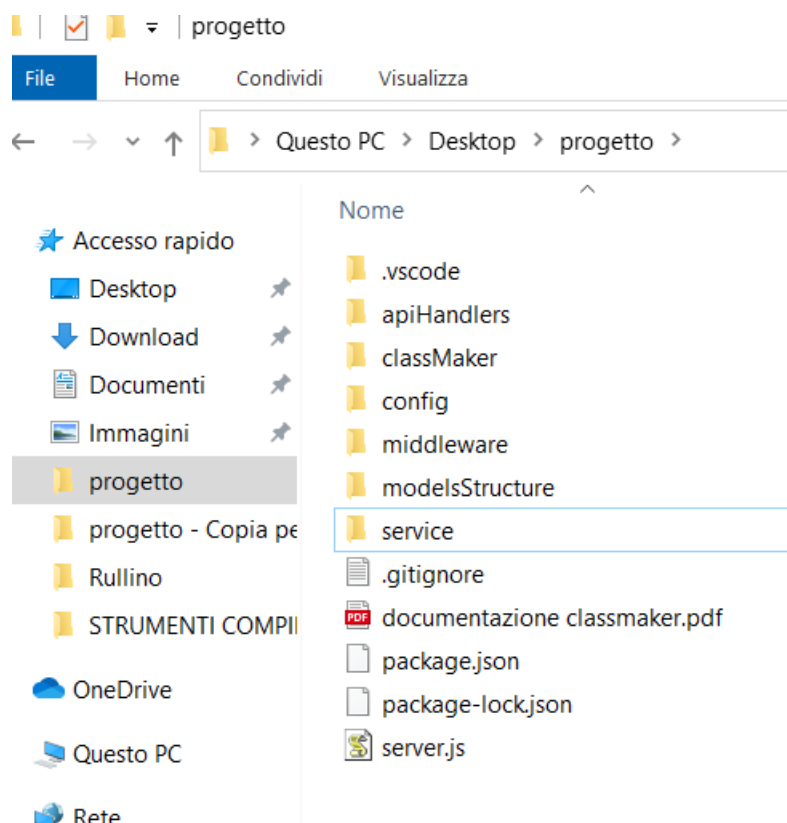
A framework for nodejs that allows us to easily describe the data model and greatly simplifies the writing of aggregation queries on mongo db.

## Requirements

- Node.js
- MongoDB

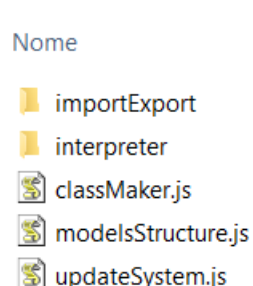
## Where to start

You have a folder with a simple sample project inside. Something like this, it may be slightly different in case of updates of the projects.



Now it's necessary to enter in the folder and run the usual command  
**npm install**

Now look at the folder **classMaker**, it's the one that interest to us.  
Inside this folder there are all the scripts of the framework.



For the moment you need to know this three things:

1. classMaker.js is a compiler of a .json file.
2. Inside importExport there is a import.js file that permit us to import data in mongoDB using csv files
3. updateSystem runs classMaker.js and then the import script.

Now return back and open the other important folder: **modelsStructure**. All other folders and files are just an example of a classmaker based project to get started using it.

This is a very important folder. Here the developer declare the structure of the project's data model. It is possible to declare it in .json or in .yaml. Yml is faster to write and more concise but it might be easier to wrong the syntax. On the other hand any developer know json. It's not really important in which format you write the model's structure. When **modelstructure.js** run it **choose to compile the last saved file (between modelsStructure.json and modelsStructure.yaml)** because the last saved file it's the last modified. If there is only a file, it choose that one. In the folder. If you open modelsStructure.json you will find something like this:

```
{
  "classes": {
    "customItem": {},
    "user": {
      "attributes": {
        "username": {
          "type": "string",
          "needed": true,
          "unique": true,
          "frontEnd": true
        },
        "password": {
          "type": "string",
          "needed": true,
          "frontEnd": false
        },
        "email": {
          "type": "string",
          "needed": true,
          "unique": true,
          "frontEnd": false
        }
      },
      "extends": "CustomItem"
    }
  }
}
```

It's a very simple example. You can see that in this example there are two classes **CustomItem** and **User**. In truth they are declared as customItem and user but the first letter of classes is considered a capital letter anyway when javascript classes are generated. You can observe that CustomItem is empty and User **extends** CustomItem. That because each classes extend item by default. Therefore

CustomItem extends Item. Item is defined in a notwritable file, you can change permissions of the file and overwrite it if you really want, but declare a CustomItem and modify it as your needs is the best practice. When you declare an attribute you must specify the type. The types that are supported for attributes are **string**, **int** (integer), **number**, **bool**, **image** (base64), **date**, **array**. When you assign a value to an attribute the value is converted (if it's possible) to the right type. For example if u set an int attribute = "3", the "3" is converted to 3. But if the conversion is impossible the attribute is assigned to his default value (**!!!è meglio così o che ignori l'assegnazione!!!???**). models/utilities/converter.js there is possible to modify it.

You can also specify if the attribute is needed ("needed": true/false), if must be **unique** ("unique":true/false), its **default** value ("default": "this is an example"). If you set a default value, if the value of the attribute would be setted undefined or null it will be setted to its default value instead. Strings hve default value "", ints have defalut value 0, arrays [].

Each class have a method frontEnd(), this method return an object Which every attribute of the original object but the attributes with "**frontEnd**" setted as false. FrontEnd method is usually used to modify the data model before send it from back-end side to front-end side. By default needed, unique are setted to false, frontEnd is setted to true, and default depends on the type.

But there are not only attributes in a class. There are also **staticAttributes** and **virtualAttributes**.

```
{
  "classes": {
    "milk": {
      "staticAttributes": {
        "expiration": {
          "type": "int",
          "value": 518400000,
          "comment": "6 days in milliesconds"
        }
      },
      "attributes": {
        "listedDate": {
          "type": "date",
          "needed": true
        }
      },
      "virtualAttributes": {
        "deadline": {
          "type": "date",
          "value": "listedDate +expiration",
          "frontEnd": true
        },
        "goneBad": {
          "type": "bool",
          "value": "deadline<new Date()"
        }
      }
    }
  }
}
```

if you want to put a description about an attribute you just write **comment**

staticAttributes are static attributes of the class. They are not saved in MognoDB but just instanced as static attributes in their javascript class. virtualAttributes appear in the class as normal attributes but they are not saved in mongodb, their value is calculated. The expression of a virtual attribute is compiled. At the moment the compiler understand expression with brackets and these other symbols, **+, -, \*, /, <, >, ==, !=, &&, ||, .length**

**The syntax of the query follow normal rules** (for example multiplication have priority respect sums etc).

It is possible to put a switch in a query. For example:

“value”: “SWITCH typeCode CASE \"A\" THEN: creationTime + additionalTime  
DEFAULT: creationTime ENDSWITCH > new Date()”

In this example the query could be creationTime+additionalTime > new Date() or creationTime > new Date() depending on the typeCode attribute of each document. So the syntax for a switch case is **SWITCH**: field\_to\_check **CASE** case1value **THEN**: case1query **DEFAULT**: defaultquery **ENDSWITCH**  
You can have all cases you want in a switch case.

In the picture above we have the virtual attribute deadline that is a sum of an attribute and a static attribute. The compiler recognize the name of attributes, staticAttributes and other virtualAttributes. The value of the virtualAttribute goneBad is “deadline < new Date()”. The compiler can recognize the word deadline and replace it with listedDate+expiration. New Date() is not recognize as an attribute of the class so it will be read it as normal javascript stuff.

After you run node ./classMaker/classMaker.js you can see that a new folder is created. The folder is models. Inside models there are two folders, interceptors and defaults. Inside default there is the file of item.js and of defaultModels.js. In DefaultModels.js there are all the javascript classes we have declared. Each class have a lots of different method we will analyze. In interceptors instead, there is a different file for each class, these files are created only if they don't exist, so if you modify an interceptor file it won't be overwritten.

```

models > interceptors > UserInterceptors.js > UserInterceptors
1  class UserInterceptors {
2    static init(user) {}
3    static async frontEnd(frontEnd, user) {
4      let picture = await Pictures.findOneBy_id(user._idProfilePicture);
5      frontEnd.profilePicture = picture.base64;
6      return frontEnd;
7    }
8    static justBeforeSave(user) {
9      return true;
10   }
11   static justAfterSave(user) {}
12 }

```

This is UserInterceptor.js. `init` it runs at the end of the constructor. `frontEnd` at the end of the `frontEnd()` method. `justBeforeSave` and `justAfterSave` run (before and after) an object is saved or updated in the DB. Probably I will add at least a fifth interceptor that run after the `delete()` method. In this class only the method `frontEnd` is customized. The methods **await** their async interceptors. **Only if they are async**, if they aren't the `async/await` is not necessary and is not written in the modelClass. The interceptors are copied and pasted with the model classes in `models/defaults/defaultModels.js`

Now if u open `defaultModels.js` you can finally see how the classes are. The file can't be manually overwritten and if you change the permissions of the file and modify stuff you will lose all your changes when you will run `classMaker/classMaker.js` again. Each class extends by default `Item` if doesn't extend already another class. The constructor of each class has only a parameter, this parameter must be an object with all the values we want assign to our instance of the class.

```

constructor(object) {
  super(object);
  if (object) {
    this.ip = object.ip;
    this.nome = object.nome;
    this.paroladordine = object.paroladordine;
    this.posta = object.posta;
    this.fotoprofilo = object.fotoprofilo;
    this.seguiti = object.seguiti;
    this.carrello = object.carrello;
  }
  UtenteInterceptors.init(this);
}

```

The attributes are declared as **private** attributes and each one have a **getter** and a **setter**. The getter give simply acces to read the value while the setter check if the value we want to assign is of the right type before do it (for example check if name

is a string). In case of a setter of a Date attribute, it accept Dates or millieconds. In case of type Image the setter check if the string in input is a base64.

```
const MilkInterceptors = require('../lib/interceptor/milkinterceptor');
class Milk extends Item {
  #listedDate;
  constructor(object) {
    super(object);
    if (object) {
      this.listedDate = object.listedDate;
    }
    MilkInterceptors.init(this);
  }

  get listedDate() {
    return this.#listedDate;
  }
  set listedDate(listedDate) {
    if (listedDate == null || listedDate == undefined)
      this.#listedDate = undefined;
    else if (listedDate instanceof Date)
      this.#listedDate = listedDate;
    else if (Number.IsInteger(listedDate))
      this.#listedDate = new Date(listedDate);
  }
}
```

Virtual attributes have getter but not setter.

```
get deadline() {
  return this.listedDate + Milk.expiration
}
get goneBad() {
  return this.listedDate + Milk.expiration < new Date()
}
```

After you have instanced an object, your instance have some important methods: **isValid()**, **frontEnd()**, **save()**, **delete()** and **getModel()**

**isValid()** check if the instance have all the unique values setted. It is a synchronous method

**frontEnd()** we have already discuss about it. It return the object we want to send to frontend side. It remove all the attributes declared with frontEnd:false. It has a very important interceptor to modify the frontEnd object. It is an asynchronous method

**save()** check if the instance is valid. If it is, the save method will get the model and save it (or just update it )in mongodb (it will be inserted if it has a new \_id or it will be modified if a document with the same \_id already exist). It is an asynchronous method

**getModel()** give access of an object with all attributes of the instance. Synchronous

**delete()** remove from mongodb the document with the same `_id` of the instance.

### Asynchronous

Suppose you have a collection of user and a collection of messages. Suppose you want when a user is removed that all his messages are removed. You can declare this in the `modelstructure.json/yml` file. There are two ways. You can specify it in the class `User` writing the list of children and the attribute that binds the two collection, or you can specify it in the class `Comment` writing the list of parent (and the attribute that binds the two collection). The attribute that binds the two collections must be in the children one. Probably the nomenclature children and parents isn't appropriate for this scenario but I've never changed it. Each class can have more than a parent and more than a children. Here two examples:

ssMaker > modelsStructure > {} modelsStructure.json > ...

```
1  {}
2
3  "classes": {
4    "user": {
5      "attributes": {
6        "name": {
7          "type": "string",
8          "needed": true,
9          "unique": true
10       },
11       "password": {
12         "type": "string",
13         "needed": true,
14         "frontend": false
15       }
16     },
17   },
18   "comment": {
19     "attributes": {
20       "_idUser": {
21         "type": "string",
22         "needed": true
23       },
24       "text": {
25         "type": "string",
26         "needed": true
27       },
28       "like": {
29         "type": "int"
30       }
31     },
32   },
33   "parents": {
34     "user": "_idUser"
35   }
36 }
37
38 }
```

ssMaker > modelsStructure > {} modelsStructure.json > ...

```
{
  "classes": {
    "user": {
      "attributes": {
        "name": {
          "type": "string",
          "needed": true,
          "unique": true
        },
        "password": {
          "type": "string",
          "needed": true,
          "frontend": false
        }
      },
      "children": {
        "comment": "_idUser"
      }
    },
    "comment": {
      "attributes": {
        "_idUser": {
          "type": "string",
          "needed": true
        },
        "text": {
          "type": "string",
          "needed": true
        },
        "like": {
          "type": "int"
        }
      }
    }
  }
}
```

When the method `save()` is used on an instance of a class that has declared at least a parent class, there is the check (for each parent) about the existence in mongodb of a page json that represent an object of the parent class with the binding attribute. If there isn't the instance can't be saved.

## MONGODB QUERIES

Now it's time to explain the queries. Each class have his static methods for queries. All this methods **return a promise**. The promise return the documents has **an array of instanced object of the class**. All find method are asynchronous

**Class.findAll**(skip, limit) => the most simple, get the documents in a collection. It has 2 params: skip (by default is 0) indicates the number of documents you want to skip; limit (by default is the max possible number) indicates the max number of documents we want.

Examples:

`User.findAll(1,100)`

**Class.findOneByUniquefield**(uniquevalue) => get the unique document with that field for that value. Here the promise return **only a object** and not an array.

Examples:

`User.findOneBy_id("dl1PalmYU")`  
`User.findOneBy_name("PincoPallino")`

**Class.findWherefield**(field, skip, limit) => *get all the documents with the right value for the field. Here as in findAll there is the possibility of skip and limit. There is a findWhere **not only** for **attributes** but **also** for **virtualAttributes**.*

Examples:

`Comment.findWhere_idUser("dl1PalmYU")`

**Class.findWhereObject**(object, skip, limit) => *get all the documents with that math with the object*



***This doesn't support virtual attributes at the moment***

And now the **most powerful and interessant** method.

***Classe.find(query, skip, limit)*** => It works as the others methods, return a promise with an array but wants a query as parameter. For the moment accept only ***&& || == !=***  
***NB*** In the == and != operation, the second element must be a value, it cannot be an attribute or a virtual attribute because mongodb does not support self join. So you have to put your attribute/virtualattribute before the == and the value after that. If you want to check that match you need to declare a virtualAttribute which does.

How works the query. There is a little interpreter that take the query written in a human way and translate it in a mongo aggregation query. The interpreter understand names of **attributes**, **virtualAttributes**. Names of static attributes are unnecessary because you can get access at their value while you declare the query string. The interpreter this time can't understand javascript. Once the interpreter has translated a query, the interpreter stores the solution in a dictionary, so the next time it immediately finds the solution. The dictionary takes into account both the query and the reference class. So different class can have the same query but still different translation (for example the two classes have a different value for static attributes or virtual attributes namesake) .This interpreter is not so powerful as that one there is when classmaker compile virtualAttributes value. That because you should declare your complicated query in virtualAttributes value. While in find method you should just do logical expression of attributes and virtualAttributes

Example:

*Comment.find('\_idUser=="dl1PalmYU" && like ==5')*

Now look at the next modelsStructure.json

```
{
  "classes": {
    "milk": {
      "staticAttributes": {
        "expiration": {
          "type": "int",
          "value": 518400000,
          "comment": "6 days in milliesconds"
        }
      },
      "attributes": {
        "listedDate": {
          "type": "date",
          "needed": true
        },
        {
          "brand": {
            "type": "string",

```

just to show how easier it is to write queries this way, I show a query using find method and how it would translate with the following models structure:

*Milk.find('goneBad==true && brand == "Parmalat"')*

```
[{"$addFields":{"goneBad":{"$lt":[{"$add":["$listedDate",518400000]},new Date()]}}},
{"$match":{"$and":[{"goneBad":true},{"brand":"Parmalat"]}}},{"$skip":0},
{"$limit":Number.MAX_SAFE_INTEGER}]
```

It is much cleaner and easier to understand and faster to write and modify.

## HYPERCLASS

Suppose that you want to have two different classes but saved in the same collection. And you want that each class have his own query methods but you want also query methods that doesn't distinct them. You need a Hyperclass. A hyperclass have attributes, staticattributes and virtualattributes. A Hyperclass have also an array named classes, that array contains the names of the classes that are affected by the hyperclass. The classes don't extend the hyperclass but they heredit their attributes, staticattributes, virtualattributes as if they virtually extend it. Normally when you use the save() method of an instance, the instance is saved in a collection of mongodb with the same name as the class of the instance. But if the class of the instance has an hyperclass, the instance is saved in the collection of the hyperclass that is named as the hyperclass but without the prefix hyper. So each class that extend the same hyperclass save their data in the same collection. Their classes have an additional attributes: typeCode. The typeCode is a string. The value of the typeCOde is the classname.toUpperCase(). So it's possible to recognize the type of documents in a collection checking their typeCode. The constructor of each class set the right typeCode by default. In defaultModels.js you have a class for each class that are included in the hyperclass. Then you have an abstract class that is extended from the other classes and another class for the hyperclass. The abstract and the hyper classes have the same name but the second have the "Hyper" prefix. The hyperClass have only static functions, the finds methods and the createFromObject(object) method. The createFromObject(object) method return an object instance of the right class that extends the hyperclass. The constructor of the hyperclass exists, it checks the typecode of the object (it has only an object as parameter as all other classes) and it returns an instance of the right class. If you use the find method of a class that affected by a hyperclass, the find gets only documents that refer that class. If you use the hyperclass find method you get each document as an instance of the class that is reported in each document's typeCode. It's very important if a class have a query of a virtualAttributes that overwrite the query of the hyperclass, the finds methods take into account the different queries. The same is if a class overwrite a staticAttribute. This time I show the yml file because is shorter.

```

hyperClasses:
  milk:
    staticAttributes:
      expiration:
        type: int
        value: 518400000
        comment: 6 days in milliesconds
    attributes:
      listedDate:
        type: date
        needed: true
    virtualAttributes:
      deadline:
        type: date
        value: listedDate +expiration
        frontEnd: true
      goneBad:
        type: bool
        value: deadline<new Date()
    classes:
      - longLifeMilk
      - freshMilk
classes:
  longLifeMilk:
    staticAttributes:
      expiration:
        type: int
        value: 15552000000
        comment: 180 days in milliesconds
  freshMilk: {}

```

In this example there is a hyperclass that have two classes. FreshMilk is an empty class so it has the typecode, heredit all attributes virtual static and nothing else. LongLifeMilk doesn't have new attributes but overwrite the expiration field, so it has the typecode, a different expiration, and heredit all others attributes. Of course the classes can have their own attributes (also virtual and static) even if they don't have any in this example. The classes affected by hyperClass are just normal classes declared as usual.

In this example the deadline attribute for longlifemilk is calcuated using his own expiration. The fresh milk instead doesn't have his own expiration so it uses the expiration of the hyperclass. So if u use for example the HyperMilk.findWhereGoneBad() it gets documets that have

typeCode="FRESHMILK" and listedDate+518400000<now

or

typeCode="LONGLIFEMILK" and listedDate+15552000000<now

## ABSTRACT CLASS

There are also abstract classes, they are like normal classes but they don't have all methods of query. If you create a new instance of the class an error is thrown. But they are useful to be extended (in this scenario no error will occur).

### CompositeKeys

It's possible to create composite keys. There are two ways, you can declare your composite keys like in the picture or you can add the property `compositeKeys` in the attribute. The composite key has a name. In this case the name is *f*. This composite key checks only two params *prefix* and *suffix* and it's declared in the class.

```
"compositeKeys": {  
  "f": {  
    "keys": [  
      "prefix",  
      "suffix"  
    ]  
  }  
}
```

The next example show an attribute that has the property `compositeKeys`. In the example below the attribute is added with the composite key "nameAndEmail" of the class

```
"email": {  
  "type": "string",  
  "needed": true,  
  "frontEnd": false,  
  "compositeKeys": "nameAndEmail"  
}
```

Of course an attribute could be part also of more composite keys. Here an example of declaration of this in the attribute:

```

    "email": {
      "type": "string",
      "needed": true,
      "frontEnd": false,
      "compositeKeys": [
        "nameAndEmail",
        "passwordAndEmail"
      ]
    }
  },

```

Here the composite keys that check email are two, so the name of the compositeKeys must be declared in an array. We know that PasswordAndEmail isn't an useful key but it's just an example.

You can also declare a composite key in the class and write in there only some attributes and add the others with the declaration in the attribute.

The compositeKeys are inherited but if the parentclass is an hyperclass. Each subclass share the same collection with the name of the hyperclass. So if you add a compositekey in a hyperclass that key is applied in the collection of all subclasses. Instead if you want to add a compositekeys that works only for a subclasses you have to declare the compositekeys in the subclasses. In this way classmaker add a filter to the compositekey checking the typecode.

```

},
"classes": {
  "customItem": {},
  "userA": {
    "attributes": {
      "prefix": {
        "type": "string",
        "compositeKeys": "f"
      },
      "suffix": {
        "type": "string"
      }
    },
    "compositeKeys": {
      "f": {
        "keys": [
          "suffix"
        ]
      }
    }
  }
}
}

```

Suppose that there is a hyperClass User that has UserA in its classes. Now I show a piece of compiledStructure.js. compiledStructure.js is a the compiled version of modelsStructure.js or modelsStructure.yml created by classMaker before it starts to write classes. How you can see, the prefix attribute is added to the keys of f, and a filter that check the typecode is added. So this compositekeys doesn't influence other subclasses of the hyperClass.


```
    },
    "compositeKeys": {
      "f": {
        "keys": [
          "suffix",
          "prefix"
        ],
        "filter": "typeCode==\"USERA\""
      }
    },
    "itsHyperClass": "User",
    "collection": "User",
    "extends": "User",
    "name": "UserA",
```

The others fields "itsHyperClass", "Collection", "extends", "name", ... are added by classMaker when it is compiling the modelsStructure.

I introduced also

**Class.findOneByObject(object)** => the method checks if there is at least a unique key or a composite key and uses it to find the document with that key. It uses the first key it finds in the object and uses only that for the query. It ignores all other fields (but typeCode if the class has an hyperClass). that fits with compositeKeys.

## IMPORT EXPORT

```
ssMaker > importExport >  prova.csv
1  #insert usera
2  username, password, email, n
3  giovanni, 1234, giovanni@example.com, 2
4  marco, 1234, marco@example.com, 3
5
6  #insert userA
7  username, password, email, n
8  anna, 1234, anna@example.com,5
9  giulia,1234,giulia@example.com,
10
11 #update userA BY email
12 username, password, email, n
13 nino, 1234, giovanni@example.com, 12
14
15 |
```

In classMaker/importExport/import.js there is a script that reads a csv and runs some instructions. At the moment it is a work in progress so it reads only the classMaker/importExport/prova.csv file. The #insert command tries to create an instance of UserA for every row (except that one of the header) and then each instance will use their save method.

The #insertupdate in this example tries to find and get (for each row but the header one) an instance of UserA using the UserA.findOneByEmail method. Then it will update each attribute with the new value. Then the instance will save itself.

If you don't specify a BY the update search with the findOneByObject. It's very handy to use but could create conflicts if there are multiple keys and they don't correspond to the document's value. In the example above (if username and email are both keys), we don't know if the findOneByObject uses the username for the search or the email (so it is better if we specify), but we can check it in the models/default/defaultModels.js!

There is also classMaker/importExport/export.js. It needs a parameter when you run it. For example classMaker/importExport/export.js user. At the moment the script gets all data it finds in the collection user and stores it as a csv in the classMaker/importExport/CSVexports/user.csv

#### **DA FARE:**

1. **pensare meglio sta cosa dell'update senza il parametro by non mi piace il fatto che fa cose a caso e può creare conflitti;**
2. **Aggiungere un remove all e un remove by e un remove senza by che funziona con lo stesso algoritmo dell'update;**
3. **fare in modo che nell'import legga i file csv presi in una cartella e che li esegua in un ordine definito;**
4. **Far sì che l'export non lavori sulle collezioni ma sulle classi e che accetta delle query per scegliere quali documenti esportare;**
5. **fare una pagina web che funga da interfaccia grafica dove poter eseguire manualmente gli import/export;**
6. **fare una interfaccia grafica che permetta di navigare comodamente tra i modelli dati salvati, modificarli e salvarli;**
7. **Rivedere il metodo delete, aggiungere gli interceptors e fare che lanci per prima cosa il metodo omonimo ereditato dalla classe estesa come fanno tutti gli altri metodi.**