UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Web-SNP

Autori
U. Buonadonna, D. Sicignani, D. Tosoni

Anno Accademico
2013/2014

Capitolo 1

Database Description

In this chapter, we will explain creation of database, from code point of view and design point of view. As explained in introduction chapter, the impl lementaion of the database is made using MongoDB, the noSQL DBMS, through mongoose, the JavaScript interface beetween the program and the database.
The approach followed by this technique allows a more efficient way of handling those objects. In fact, using mongoose's schemas to define objects of the
database, permits to treat database models like any JavaScript object. To achieve this, MongoDB uses documents, a set of key-value pairs, stored in
collections, held by a database.
In the following part, we will see objects that form our database model.

1.1 Mongoose's syntax

To realize an object using mongoose, we need to follow a specific syntax.
First of all, you must place all model files related to the diagrams of the model in a specific folder within the project. Each of these files must be placed in a specific folder within the project /server/models/.// At the beginning of each file, we need to inform the database of the fact that any values passed to the builders of the models, which are not present in the schema should not be persisted. We do this by defining a string
`'use strict'`
. then you must specify the modules dependencies required to create the schema. In

our case, we must satisfy the dependency on the modules (bold type) 'mongoose' and 'crypto (a module for encrypting information.)

We do this through the following code:

```
var mongoose = require('mongoose'),
Schema = mongoose.Schema,
crypto = require('crypto');
```

At this point you switch to define the schema itself, which consists of all the necessary components for the definition of the object in question, with the attributes and relationships. Note that mongo automatically adds an ID fileld for each schema defined, and assigns it following an internal policy. The syntax for doing this is explicated in the example, where it is supposed to want to model a user with an attribute **name** of type string:

```

UserSchema = new Schema ({
  // attribute
  name: {
         type: String,
         required: true,
         validate: [validatePresenceOf, 'Name can not be blank']
     },

    // relationship
   patiens: [{
   type: Schema.Type.ObjectId,
   ref: 'Patient'
   }]
     );
```

Next, you can define a number of additional features to our model, such as

- Validations: functions called to validate a set of attributes or data specified by the programmer
- Virtuals: virtual attributes, ie not persisted

- Pre-save hooks: functions called at the time immediately before the model is saved to the database
- Methods: all methods necessary for the operation of our model.

Finally, one must specify the actual name of the model from which the schema specified as previously shown, in addition to its creation proocedere real. We do this through the following code:

```
1  mongoose.model ('User', UserSchema);
```

1.2 Model Schemas

1.2.1 Family

A Family represents a set of one or more patients (modeled in patient schema). It can be created only by an user with admin privileges. This schema allow to group patients with similar genomic mutations as well as real families (father, mother, son ..).
Here it is the code to implement the family object:

```
1  'use strict';
2
3  /**
4   * Module dependencies.
5   */
6  var mongoose = require('mongoose'),
7      Schema = mongoose.Schema,
8      crypto = require('crypto');
9
10
11 /* Family schema */
12 var FamilySchema = new Schema({
13
14     name: {
15         type: String,
16         required: true,
17         validate: [validatePresenceOf, 'Name cannot be blank']
18     },
```

```
19
20      //Relationship
21      patiens: [{
22        type: Schema.Type.ObjectId,
23        ref: 'Patient'
24      }]
25
26  });
27
28
29  /**
30   * Validations
31   */
32  // nothing for now
33
34  /**
35   * Virtuals
36   */
37
38  // no not-persisted attributes
39  /*
40   * Pre-save hook
41   */
42  //still nothing
43
44  /**
45   * Methods
46   */
47  // no method required here. Query class
48
49
50  mongoose.model('Family', FamilySchema);
```

Attributes

- **name:** it describes family's name

Relationships

- **patients:** links family to its members

1.2.2 Variants

The table variants is what is the key concept that we want to model in our project, namely the real genomic variants; This table will always be accessed in fact during the consultation phase, where the biologist can search or include such variations, which will be stored in the table. The database is connected to almost all the tables, the fact that testifies to the centrality of his role.

To implement that table we use the following code:

```
'use strict';

/**
 * Module dependencies.
 */
var mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    crypto = require('crypto');


/* Variant schema */
var VariantSchema = new Schema({
    chr: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'CHR cannot be blank']
    },
    start: {
        type: String,
        required: true,
```

```
21          validate: [validatePresenceOf, 'start cannot be blank']
22      },
23      end: {
24          type: String,
25          required: true,
26          validate: [validatePresenceOf, 'End cannot be blank']
27      },
28      ref: {
29          type: String,
30          required: true,
31          validate: [validatePresenceOf, 'Ref cannot be blank']
32      },
33      alt: {
34          type: String,
35          required: true,
36          validate: [validatePresenceOf, 'Alt cannot be blank']
37      },
38
39
40      // Here start relations
41      gene: {
42          type: Schema.Types.ObjectId,
43          ref: 'Gene'
44      },
45
46      pathogenicity: {
47          type: Schema.Types.ObjectId,
48          ref: 'Pathogenicity'
49      },
50
51      patients: [{
52          type: Schema.Types.ObjectId,
53          ref: 'Patient'
54      }],
55
56      dbSNPs: [{
57          type: Schema.Types.ObjectId,
58          ref: 'DbSNP'
```

```
59      }],
60
61      esps: [{
62          type: Schema.Types.ObjectId,
63          ref: 'esp'
64      }],
65
66      sequencings: [{
67          type: Schema.Types.ObjectId,
68          ref: 'Sequencing'
69      }]
70
71  });
72
73
74  /**
75   * Validations
76   */
77  // nothing for now
78
79  /**
80   * Virtuals
81   */
82
83  // no not-persisted attributes
84  /*
85   * Pre-save hook
86   */
87  //still nothing
88
89  /**
90   * Methods
91   */
92  // no method required here. Query class
93
94
95  mongoose.model('Variant', VariantSchema);
```

Attributes

- **chr:** it's a five-characters string that //.....NON E FINITO
- **start,end:** integers that describe the coordinate of the mutation within the chromosome; if they match, it means that we are dealing with a point mutation
- **ref:** represents the correct basis, that is what should be if there were no mutation
- **alt:** represents what we find is actually due to the mutation; one. or - represent a deletion, while a sequence, for example CTTG ..., represents an insertio

Relationships

- **gene:** links the variant with the gene affected by the change
- **pathogenicity:** describes the variant pathogenicity, that is if the variant cause diseases
- **patients:** links patients affected by that varant
- **dbSNPs:** connects the variant to its dbSNPs
- **esps:** BOH - YEAH
- **sequencings:** all sequencings affected by this variation