UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

SNP Web-portal

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

Anno Accademico
2013/2014

Capitolo 1

Database Description

In this chapter we will explain the creation of the database, from code point of view as well as from design point of view.
The **database** is the core part of the back end of the application, because it is the structure that keeps stored all the data necessary for the operation of the portal. Because of its importance, we spent much of our time to decide its structure (tables, relationships, multiplicity, ...) and only after a thorough analysis of results we moved to implementation.

As explained in the introductory chapter, the software implementation of database was performed using **MongoDB** through **Mongoose** (a MongoDB object modeling tool). MongoDB is an open-source document database, and the leading NoSQL database. It is based on *Document-Oriented Storage* and unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections do not enforce document structure. This flexibility facilitates the mapping of documents to an entity or an object.
Mongoose helps to have a more efficient management of the objects; in fact, using mongoose's schemas to define database's objects, permits to treat database's models as commons JavaScript's objects.

Throughout the chapter, we will see the objects that compose the model of our da-

tabase, but, first of all, let us focus on the syntax used to communicate with MongoDB and Mongoose.

1.1 Mongoose's syntax

To realize an object using mongoose, we need to follow a specific syntax.
First of all, we should place all files related to the models (one file for each model) in a specific folder of the project; this folder is as follows:

/server/models

At the beginning of each file, we have to inform the database that all data passed to builders, but that are not present in the schema that we will define below, must not be stored.
To do this we use the following line of code:

```
1                'use strict';
2
```

◇

Next, we must specify needed *modules dependencies*. In our project, we have to insert the dependency from mongoose and crypto (a module for encryption of information):

```
1                var mongoose = require('mongoose'),
2                Schema = mongoose.Schema,
3                crypto = require('crypto');
4
```

◇

Now we move to define the *schema*, specifying all necessary components (attributes and relationships). Note that MongoDB will automatically add an ID field to each schema and assigns it using an internal policy.

For example, if we want to model a *User* with an attribute *name* of type *string*:

⟨ *Example schema ?* ⟩ ≡

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

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

```

◇

Fragment never referenced.

Next, we 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, not persisted
- **Pre-save hooks**: functions called at the time immediately before the model is saved to the database
- **Methods**: all methods needed for the operations on our model

Finally, we must specify the actual name of the model described by the schema specified as previously shown:

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

1.2 Model Schemas

These are the model schemas that compose our database.

1.2.1 Variants

The model *Variant* represents the key concept that we want to represent in our work: **genomic variants**.
This table will always be accessed during any kind of consultation, in fact are variants what biologists are looking for or insert through the portal.
The centrality of the concept and, therefore, of the table is also demonstrated by the fact that in the database schema (see chapter *Introduction*) Variant table is linked to almost all the other tables.

Let us look at the components:

Attributes

⟨ *Variant's attributes ?* ⟩ ≡

```
var VariantSchema = new Schema({
    chr: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'CHR cannot be blank']
    },
    start: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'start cannot be blank']
    },
    end: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'End cannot be blank']
    },
    ref: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Ref cannot be blank']
    },
    alt: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Alt cannot be blank']
    },
```

◇

Fragment never referenced.

- **chr**: it is a five-character string that ... Mazza's help needed...

- **start, end**: integers that describe the coordinate of the mutation within the chromosome; if they match, it means that we have a punctiform mutation
- **ref**: represents the correct bases, as they would be if there were no mutation
- **alt**: represents what we actually find due to the mutation; a . or a - represent a deletion, while a sequence, for example CTTG..., represents an insertion

Relationships

- **gene**: links the variant with the gene affected by the mutation
- **pathogenicity**: describes the variant pathogenicity, that is if the variant cause diseases or not
- **patients**: links to patients affected by this varant
- **dbSNPs**: connects the variant to its dbSNPs
- **esps**: ... **Mazza's help needed...**
- **sequencings**: all sequencings affected by this variation

1.2.2 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);
```

Let us look at the components:

Attributes

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

Relationships

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