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 code components:

Attributes

⟨ *Variant's attributes ?* ⟩ ≡

```
/* Variant schema */
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 describing the chromosome affected by the variation

- **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

⟨ *Variant's relationships ?* ⟩ ≡

```
    // Here start relations
    gene: {
        type: Schema.Types.ObjectId,
        ref: 'Gene'
    },

    pathogenicity: {
        type: Schema.Types.ObjectId,
        ref: 'Pathogenicity'
    },

    patients: [{
        type: Schema.Types.ObjectId,
        ref: 'Patient'
    }],

    dbSNPs: [{
        type: Schema.Types.ObjectId,
        ref: 'DbSNP'
    }],

    esps: [{
        type: Schema.Types.ObjectId,
        ref: 'esp'
    }],

    sequencings: [{
        type: Schema.Types.ObjectId,
        ref: 'Sequencing'
    }]

});
```

◇

Fragment never referenced.

- **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**: Exome Sequencing Project
- **sequencings**: all sequencings affected by this variation

1.2.2 dbSNP

dbSNP is the key concept of the database, that is the **Single Nucleotide Polymorphism Database** (a free public archive for genetic variation within and across different species developed and hosted by the *National Center for Biotechnology Information (NCBI)* in collaboration with the *National Human Genome Research Institute (NHGRI)*).

Let us look at the code components:

Attributes

⟨ *dbSNP's attributes ?* ⟩ ≡

```
/* DbSNP schema */
var DbSNPSchema = new Schema({
    dbSNP: {
        //key
        type: String,
        required: true,
        unique: true,
        validate: [validatePresenceOf, 'DbSNP cannot be blank']
    },
    freqAlt: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'freqAlt cannot be blank']
    },
    freqRef: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'freqRef cannot be blank']
    },
```

◇

Fragment never referenced.

- **dbSNP**: identifier of the variation in the dbSNP database
- **freqAlt**: frequency of the alternative allele (variant) in the population 1000 Genomes
- **freqRef**: frequency of the reference allele

Relationships

⟨ *dbSNP's relationships ?* ⟩ ≡

```
    //Relationship
    variants: [{
        type: Schema.Type.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

- **variants**: links dbSNP to its variants

1.2.3 Gene

A *Gene* is the molecular unit of heredity of a living organism. Genes hold the information to build and maintain an organism's cells and pass genetic traits to offspring. All organisms have genes corresponding to various biological traits, some of which are instantly visible, such as eye color or number of limbs, and some of which are not, such as blood type, increased risk for specific diseases, or the thousands of basic biochemical processes that comprise life.

Let us look at the code components:

Attributes

⟨ *Gene's attributes ?* ⟩ ≡

```
/* Gene schema */
var GeneSchema = new Schema({
    gene: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Gene cannot be blank']
    },
    region: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Region cannot be blank']
    },
    mutation: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Mutation cannot be blank']
    },
    annotation: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Annotation cannot be blank']
    },
```

◇

Fragment never referenced.

- **gene**: identifier of the gene
- **region**: the region in which the gene is located
- **mutation**: the mutation that affects the gene
- **annotation**: any annotations

Relationships

⟨ *Gene's relationships ?* ⟩ ≡

```
    // Relationship
    variants: [{
        type: Schema.Type.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

- **variants**: links gene to its variants

1.2.4 Pathogenicity

Pathogenicity indicates whether a mutation is pathogenic (able to create damages to the organism) or not.

Let us look at the code components:

Attributes

⟨ *Pathogenicity's attributes ?* ⟩ ≡

```
/* Pathogenicity schema */
var PathogenicitySchema = new Schema({
    SIFT: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'SIFT cannot be blank']
    },

    polyPhen: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'polyPhen cannot be blank']
    },

    mutationTaster: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'mutationTaster cannot be
    blank']
    },

    mutationAssessor: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'mutationAssessor cannot be
    blank']
    },

    GERPpp: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'GERPpp cannot be blank']
    },

    pyoloP: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'pyoloP cannot be blank']
    },

    SiPhy: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'SiPhy cannot be blank']
    },
```

- **SIFT, polyPhen, mutationTaster, mutationAssessor**: all four are predictors of pathogenicity of the mutation. The numbers are the scores of the pathogenicity.
- **GERPpp, pyoloP, SiPhy**: all three are predictors of conservation of the mutation. The numbers are the scores of conservation.

Relationships

⟨ *Pathogenicity's relationships ?* ⟩ ≡

```
    //Relationship
    variant: {
        type: Schema.Type.ObjectId,
        ref: 'Variant'
    }

});
```

◇

Fragment never referenced.

- **variant**: links pathogenicity to its variant

1.2.5 Sequencing

DNA Sequencing is the process of determining the nucleotide order of a given DNA fragment. The sequence of DNA encodes the necessary information for living things to survive and reproduce, so determining the sequence is therefore useful in fundamental research into why and how organisms live, as well as in applied subjects.

Let us look at the code components:

Attributes

⟨ *Sequencing's attributes ?* ⟩ ≡

```
/* Sequencing schema */
var SequencingSchema = new Schema({
    //patient id and date are keys
    patientId: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'patient cannot be blank']
    },
    date: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'date cannot be blank']
    },
    patientHealthStatus: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'patientHealthStatus cannot be
     blank']
    },
    sequencerName: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'sequencerName cannot be blank
    ']
    },
    sequencerModel: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'sequencerModel cannot be
    blank']
    },
    referenceGenome: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'referenceGenome cannot be
    blank']
    },

    detail :{
        qual: {
            type: String,
            required: true,
            validate: [validatePresenceOf, 'qual cannot be blank']
        },
        filter: {
```

- **patientId**: identifier of the patient.
- **date**: the date when sequencing was performed.
- **patientHealthStatus**: if the patient is diseased or not.
- **sequencerName**: name of the sequencer.
- **sequencerModel**: model of the sequencer.
- **referenceGenome**: the reference genome.
- **qual**: describes the quality of the sequencing.
- **filter**: reference allele.
- **genotype**
- **genotypeQuality**: quality of the genotype.
- **readsDepth**: the depth of the reads.
- **ref**: string that indicates whether the quality of the variant is good (PASS).
- **altFilterReads**: number of reads that do not contain the mutation to their inside.
- **genotypeLikelihood**: probability that the determined genotype is correct.
- **haplotypeScore**: indicate the presence of misaligned reads in the neighborhood of the variant.
- **strandBias**: all four are predictors of pathogenicity of the mutation.

Relationships

⟨ *Sequencing's relationships ?* ⟩ ≡

```
        },
        genotype: {
            type: String,
            required: true,
            validate: [validatePresenceOf, 'genotype cannot be blank']
        },
        genotypeQuality: {
```

◇

Fragment never referenced.

- **variants**: links sequencing to its variants

1.2.6 ESP

ESP is the part of the database that represents locally the interesting portion of the database of the **Exome Sequencings Project**, the goal of which is to discover novel genes and mechanisms contributing to heart, lung and blood disorders by pioneering the application of next-generation sequencing of the protein coding regions of the human genome across diverse, richly-phenotyped populations and to share these datasets and findings with the scientific community to extend and enrich the diagnosis, management and treatment of heart, lung and blood disorders.

Let us look at the code components:

Attributes

⟨ *esp's attributes ?* ⟩ ≡

```
/* Esp schema */
var EspSchema = new Schema({
    ESP6500_ALL: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'ESP6500_ALL cannot be blank']
    },
    ESP6500_AA: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'ESP6500AA cannot be blank']
    },
    ESP6500_EA: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'ESP6500_EA cannot be blank']
    },
```

◇

Fragment never referenced.

- **ESP6500_ALL**: frequency of the mutation in the EVS database and in the overall population
- **ESP6500_AA**: frequency of the mutation in the EVS database and in the American/African population
- **ESP6500_EA**: frequency of the mutation in the EVS database and in the European /American population

Relationships

⟨ *esp's relationships ?* ⟩ ≡

```
    //Relationship
    variants: [{
        type: Schema.Type.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

- **variants**: links esp to its variants

1.2.7 Patient

A *Patient* describes a patient in the real world. Can only be created by a user with Admin privileges.

Let us look at the code components:

Attributes The Patient object has no attributes to be specified; it only has an identifier (**ID**)that is added from MongoDB by default.

Relationships

⟨ *Patient's relationships ?* ⟩ ≡

```
/* Patient schema */
var PatientSchema = new Schema({

  //Relationship
  variants: [{
    type: Schema.Type.ObjectId,
    ref: 'Variant'
  }],

  family: {
    type: Schema.Type.ObjectId,
    ref: 'Family'
  }

});
```

◇

Fragment never referenced.

- **variants**: links patient to its variants
- **family**: links patient to its family

1.2.8 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...).

Let us look at the code components:

Attributes

⟨ *Family's attributes ?* ⟩ ≡

```
/* Family schema */
var FamilySchema = new Schema({

    name: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Name cannot be blank']
    },
```

◇

Fragment never referenced.

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

Relationships

⟨ *Family's relationships ?* ⟩ ≡

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

});
```

◇

Fragment never referenced.

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

1.2.9 User

The *User* object describes a user of the system, a *biologist* who accesses the portal to carry out one of the supported operations (described in the introductory chapter).

Let us look at the code components:

Attributes

⟨ *User's attributes ?* ⟩ ≡

```
/**
 * User Schema
 */
var UserSchema = new Schema({
    name: {
        type: String,
        required: true,
        validate: [validatePresenceOf, 'Name cannot be blank']
    },
    email: {
        type: String,
        required: true,
        match: [/.+\@.+\..+/, 'Please enter a valid email'],
        validate: [validatePresenceOf, 'Email cannot be blank']
    },
    username: {
        type: String,
        unique: true,
        validate: [validatePresenceOf, 'Username cannot be blank']
    },
    roles: {
        type: Array,
        default: ['authenticated']
    },
    hashed_password: {
        type: String,
        validate: [validatePresenceOf, 'Password cannot be blank']
    },
    provider: {
        type: String,
        default: 'local'
    },
    salt: String,
    facebook: {},
    twitter: {},
    github: {},
    google: {},
    linkedin: {}
});
```

◇

Fragment never referenced.

These attributes are all self-explanatory.

Methods

⟨ *User's validation ?* ⟩ ≡

```
/**
 * Validations
 */
var validatePresenceOf = function(value) {
    // If you are authenticating by any of the oauth strategies, don't validate.
    return (this.provider && this.provider !== 'local') || value.length;
};
```

◇

Fragment never referenced.

⟨ *User's methods ?* ⟩ ≡

```
/**
 * Virtuals
 */
UserSchema.virtual('password').set(function(password) {
    this._password = password;
    this.salt = this.makeSalt();
    this.hashed_password = this.hashPassword(password);
}).get(function() {
    return this._password;
});

/**
 * Pre-save hook
 */
UserSchema.pre('save', function(next) {
    if (this.isNew && this.provider === 'local' && this.password && !
    this.password.length)
        return next(new Error('Invalid password'));
    next();
});

/**
 * Methods
 */
UserSchema.methods = {

    /**
     * HasRole - check if the user has required role
     *
     * @param {String} plainText
     * @return {Boolean}
     * @api public
     */
    hasRole: function(role) {
        var roles = this.roles;
        return roles.indexOf('admin') !== -1 || roles.indexOf(role)
    !== -1;
    },

    /**
     * IsAdmin - check if the user is an administrator
     *
     * @return {Boolean}
     * @api public
     */
```

Functions performed by these methods are explained in the code comments.