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

Introduction

In this chapter it is provided a brief introduction about the project described in the context of this essay.

1.1 Medical Background

The project focuses on the concept of **SNP**. SNP, *Single Nucleotide Polymorphism*, is defined as a *DNA sequence variation* occurring when a Single Nucleotide — A, T, C or G — in the genome (or other shared sequence) differs between members of a biological species or paired chromosomes. These variations can either be pathogenic, causing diseases, or completely harmless.
The main purpose for which the project is carried out is to *help biologists understanding, having an ad-hoc organized database, when a particular SNP can cause diseases or not.* The Web Portal allows them to **store** and **retrieve** Single Nucleotide Polymorphism genomics variants from a common database. This portal will then be used by biologists for this task, so it is realized so as to provide an intuitive and functional interface, which will act as an intermediary between the biologist and the actual database, facilitating their work as much as possible.

The main functions of the portal are divided into two groups, depending on the privilege level of the user who use the system at the time:

Super User

The *Super User* is the user with the *highest action privilege*, in fact it is in full possession of all the functionalities of the system.
As such, it can:

- create a "family" with one or more members
- authorize users
- modify inserted values

Basically, it can add, modify and enter the values it wants to. Obviously, the Super User can perform all the activities that Authorized User can do.

Authorized User

The *Authorized User* is the one with a *lower degree of privilege*, so it can not perform all the typical actions of the Super User, such as edit and insert.
It only can:

- search for a patient
- search for a gene
- search for a mutation
- ...

The authorized user is someone who can query the portal, but who can not modify any values.

The distinction between types of users has been designed to *provide a control mechanism against unwanted anomalies*; in fact, if a biologist is interested in making simple queries to the system, he is able do it in a way that no changes or accidental deletions occurs.

1.2 Project Structure

We will now describe the structure of the project.

The project is divided into three parts, which represent the basic structure of the portal:

1. **Database**
2. **Database - Website interaction**
3. **Website (User Interface)**

1.2.1 Database

The database is the most important part of the whole project, in fact the portal acts as an *intermediary* between the database itself and the biologist. **All informations about the SNPs are stored in the database**. It follows that a poorly designed database can invalidate all the work related to the portal; therefore the good output of the same is conditioned first of all by a good design of the database.

We will not focus on a detailed description of the structure of the database (that will be explained in *Chapter 1*).
We can see, however, that the most important part of the database is represented by the concept of **variation**, which represents nothing more that *a variation within the genome*. All other elements that need to be managed refer to the variant; we mention by way of example the gene relative to the variant, or the patient on which refers the same variant.

1.2.2 Database - Website interaction

Interaction between Database and Website is based on a *series of queries*, that the user can send to the system to retrieve the information it wants. These queries are then executed on the database and the result returned to the user.

Items covered by those queries may be different; for example, the system allows searching for:

- patient's SNPs
- gene's SNPs
- region's SNPs
- all SNPs with certain Mutation, Genotype, Freq alt,...
- patients with same SNP or Genotype
- SNPs within a genomic region
- specific SNP
- ...

The interactions can be carried out, as explained earlier, by all users (either Autenticated or Super User). The interactions for editing and adding data may relate to any entity in the database.

1.2.3 Website (User Interface)

The portal interface is represented by web pages, containing forms allowing to submit queries.
The fundamental characteristics that this interface must have are directly related to the it's purpose; we have to make sure that the biologists are facilitated as much as possible in their work of storing and retrieving information, to ensure that their work is as fast as possible. This affects also the intuitiveness and ease of use of the interface, that have to be taken into primary consideration.

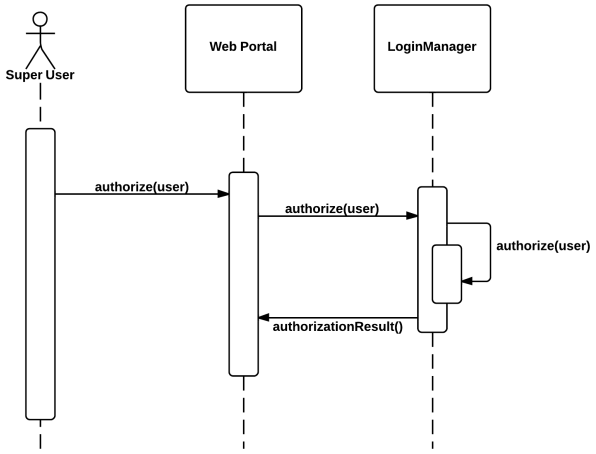
1.3 Use cases

In this section we will discuss the use cases, illustrating the use of the system.
They will show how the user interacts with the system, from the point of view of the

messages exchanged with it, and how the system handles these messages. You can see that the patterns of the use cases are similar to each other, since all gets the message from the user, and run it to the database by sending a special message for each request submitted.

We will now analyse briefly each use case.

1.3.1 Super User authorizes an User



In this use case the actors are the **Super User**, the **Web Portal** and the **Login manager**.The user's intention is to allow another user to have privilege for using the portal.

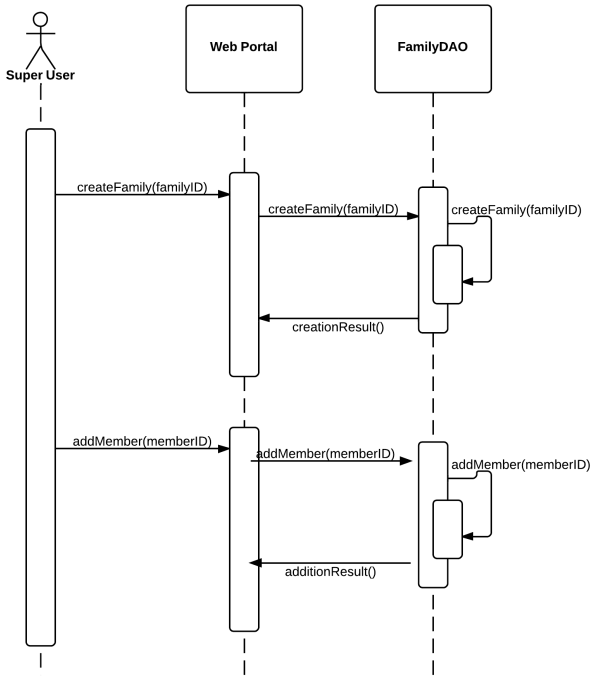
The interaction begins with the super user that sends a message *authorize(user)*to the Web portal. The latter sends the same message to the login manager, wich will be responsible to communicate with the database to record the granted permission. Successicely, the login manager sends a message to the web portal *authorization(result)*, containing the result of the operation, so that the Web Portal can know whether the authorization was granted or not.

1.3.2 Super User creates/populates a family

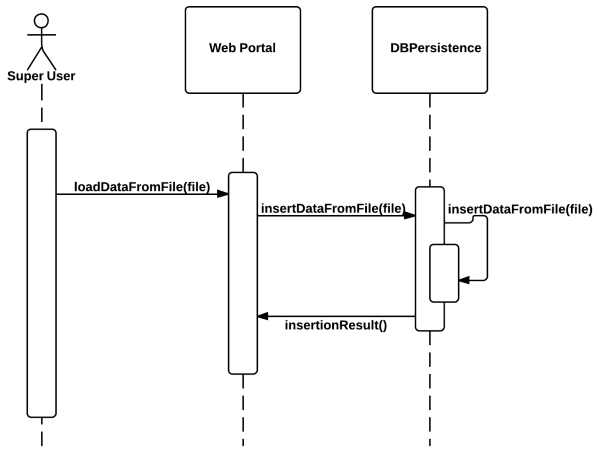
In this use case the actors are the **Super User**, the **Web Portal** and the **Family DAO**. The user's intention is to create a new family, and populate it with a member.

The interaction starts with the user sending a message *createFamily (familyID)* to Web Portal, which sends a message similar to FamilyDAO, the class responsible for managing / creating families and its interaction with the database. This class then communicates with the database itself, and sends a message containing the create result to the Web Portal.

If the family creation is successful, then the user can populate it, through a message *addMember(memberID)* sent to the Web Portal. The portal then dispatches the message to the FamilyDAO, who will add the member to the family, and returns the result to the Web Portal.



1.3.3 Super User loads data



In this use case the actors are the **Super User**, the **Web Portal** and the **DBpersistence**. The user's intention is to load the data that have to be stored in the database, for later querying. Note that data is submitted to the system by using a comma-separated file.

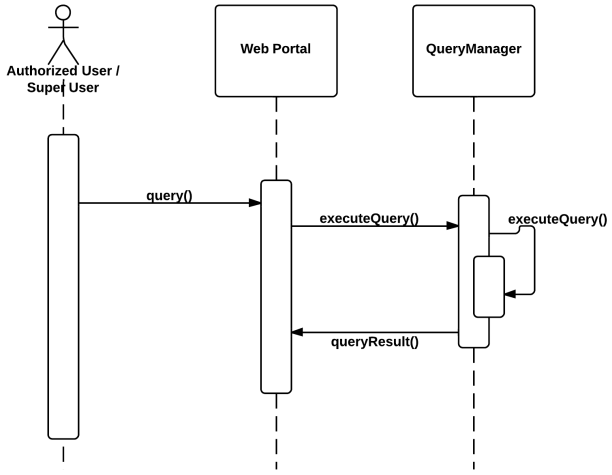
To accomplish this task, the user sends a message to the web portal *lodaDataFromFile(file)*, which file, as mentioned before, is a comma-separated file containing all the necessary data. Then the web portal sends message and data to the DBperistence class that both parse the file, and pass all information contained in file to the database, which will then store them.

Successively, this class will send the result of this insertion to the web portal through a message *insertionResult()*.

1.3.4 Authorized User executes a query

In this use case the actors are the **Authorized User**, the **Web Portal** and the **QueryManager**. The user's intention is to run a query on the system, in order to obtain the information he wants to.

The user then sends a message *query()* to the web portal, which will contain a generic query submitted by the user.The portal will send a message *executeQuery()* to QueryManager, who will actually run the query in the database, and return the results to the portal. It does this through a message *QueryResult()*



1.4 Software architecture and tecnologies

In this section we will discuss the software technologies used in the project, both for the front-end and back-end.

1.4.1 Programming language

The only programming language used throughout the project is **JavaScript**. This is because we want to use frameworks supporting the JavaScript language (that can be considered one of the most utilised languages relating to dynamic content in web pages) that ensure a great rapidity of coding.

1.4.2 Technologies

Technologies used all ri around chosen language. In particular:

Database: MongoDB

Website: AngularJS

Model: NodeJS

Framework: ExpressJS

MongoDB is a *NoSQL object-oriented DBMS*, which is based on the use of documents to represent objects. AngularJS allows to *extend HTML with instructions for building dynamic web pages*. NodeJS is an *event-driven, non-blocking network platform for building applications*. ExpressJS is a *web application framework for NodeJS*, that extends some of its features, to make development faster.

These four components form the so-called **MEAN** stack; it has been chosen for its scalability and for the quickness of development it can offer.

Capitolo 2

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.

2.1 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);
```

2.2 Model Schemas

These are the model schemas that compose our database.

2.2.1 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'
    }],

    variantDetails: [{
        type: Schema.Types.ObjectId,
        ref: 'VaiantDetail'
    }]

});
```

◇

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

2.2.2 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.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

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

2.2.3 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.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

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

2.2.4 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.ObjectId,
        ref: 'Variant'
    }

});
```

◇

Fragment never referenced.

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

2.2.5 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

2.2.6 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.ObjectId,
        ref: 'Variant'
    }]

});
```

◇

Fragment never referenced.

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

2.2.7 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.ObjectId,
    ref: 'Variant'
  }],

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

});
```

◇

Fragment never referenced.

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

2.2.8 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.ObjectId,
      ref: 'Patient'
    }]

});
```

◇

Fragment never referenced.

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

2.2.9 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.

Capitolo 3

Backend-Frontend communication

In this chapter we will explain the realization of the interaction between application's Backend and Frontend.
The interaction occurs through the implementation of **RESTful APIs that allow to interact with a MongoDB instance**.

3.1 The idea

We built some APIs (one for each item that the application handles) that:

- Handle CRUD for the item
- Use the proper HTTP verbs to make it RESTful (GET, POST, PUT, and DELETE)
- Return JSON data
- Log all requests to the console

All of this is pretty standard for RESTful APIs.

3.2 Implementation

To realize APIs, we needed to:

1. define our Node packages
2. define our models
3. declare our routes using Express
4. test our API

3.2.1 Node packages

Node Packages are the list of the support packages that are used for the realization of the application; therefore it also contains the packages needed for the APIs.

This is our package.json file:

⟨ *package.json ?* ⟩ ≡

```
{
  "name": "mean",
  "description": "MEAN.io: A fullstack javascript platformed powerd
    by MongoDB, ExpressJS, AngularJS, NodeJS.",
  "version": "0.3.3",
  "private": false,
  "author": "Linnovate <mean@linnovate.net>",
  "contributors": [
    { "name": "Lior Kesos", "mail": "lior@linnovate.net" },
    { "name": "Yonatan Ellman", "mail": "yonatan@linnovate.net" },
    { "name": "Ehud Shahak", "mail": "ehud@linnovate.net" },
    { "name": "Amos Haviv", "mail": "mail@amoshaviv" },
    { "name": "Drew Fyock", "mail": "drew@steelbisondev.com" }
  ],
  "mean": "0.3.3",
  "repository": {
    "type": "git",
    "url": "https://github.com/linnovate/mean.git"
  },
  "engines": {
    "node": "0.10.x",
    "npm": "1.3.x"
  },
  "scripts": {
    "start": "node node_modules/grunt-cli/bin/grunt",
    "test": "node node_modules/grunt-cli/bin/grunt test",
    "postinstall": "node node_modules/bower/bin/bower install"
  },
  "dependencies": {
    "assetmanager": "0.1.2",
    "body-parser": "^1.2.0",
    "bower": "^1.3.3",
    "compression": "^1.0.1",
    "connect-flash": "^0.1.1",
    "consolidate": "^0.10.0",
    "cookie-parser": "^1.1.0",
    "dependable": "0.2.5",
    "errorhandler": "^1.0.0",
    "express": "^4.2.0",
    "express-session": "^1.1.0",
    "express-validator": "^2.1.1",
    "forever": "^0.11.1",
    "grunt-cli": "^0.1.13",
    "grunt-concurrent": "^0.5.0",
    "grunt-contrib-clean": "^0.5.0",
```

In our selection of packages, those necessary for the implementation of the APIs are:

- **express**, the Node framework
- **mongoose**, the ORM we used to communicate with our MongoDB database
- **body-parser**, to pull POST content from our HTTP request

3.2.2 Model

Models required for the API are the same as those described in *Chapter 2*, to which was added an additional model: **User**. It is necessary to be able to properly handle user information as well as the levels of privilege that a user owns.

This is the User model:

⟨ *package.json ?* ⟩ ≡

```
'use strict';

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

/**
 * 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;
};

/**
 * 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,
```

3.2.3 Routes

We used an instance of the *Express Router* to handle all of our routes.
Here is an overview of the routes we made, what they do, and the HTTP Verb used to access it. *"model"* represent one of the models of our application.

Route	HTTP Verb	Description
/api/model	GET	Get all the "model"
/api/model	POST	Create a "model"
/api/model/:model_id	GET	Get a single "model"
/api/model/:model_id	POST	Update a "model" with new info
/api/model/:model_id	DELETE	Delete a "model"

To achieve this, we used the package **MERS** (*Mongoose Express Rest Service*): it is a plugin for express to expose mongoose finders as simple crud/rest operations.

3.2.4 Tests