Desenvolvimento Baseado em Modelos - Cinema

Manual Técnico

Índice

Introdução

Este manual tem como foco a documentação técnica da implementação de um programa no qual tem como objetivo a criação de um site de filmes, onde é possível pesquisar, ver detalhes, poster e até um trailer do mesmo.
Através da criação de modelos o site é construído à medida do utilizador.

De forma a atingir estes objetivos foi utilizado:

- **Metamodelação** utilizando *JSON Schema*, uma síntaxe específica de *JSON*;
- **Transformações** modelo-para-texto utilizando a linguagem *Mustache*;
- **Base de dados**: SQLite3;
- **Principais linguagens de programação**: *JavaScript* e *NodeJS*.

Depois de serem criados modelos, são geradas classes *JavaScript* que representam entidades respectivas também geradas na base de dados. Sendo criado também uma API RESTful na qual é utilizada para operações CRUD.

Modelo: Uma entidade do domínio do problema, com atributos específicos que necessitam de ser especificados. Por exemplo, é necessário definir um modelo que representará um objeto na aplicação.

Através do tema dado foi desenvolvida uma aplicação que permite a gestão de vários filmes e respectivas informações.

Metamodelação

Metamodelação: Um modelo que cria modelos.

Para a metamodelação do problema apresentado, foi utilizado **JSON Schema**, uma vertente de *JSON* que permite definir o nome, descrição, atributos de um modelo, que são chamadas de propriedades e relações com outros modelos, que se designam como referências.

O Exemplo de um modelo seguinte fornecido é a representação de um **Movie**:

```
 {
 "title": "Movie",
```

```
    "description": "Movie description",
    "type":"object",
    "properties": {
    "name":{
    "description": "movie name",
    "type":"string",
    "regex":"[a-zA-Z]+"
    },
    "year":{
    "description": "movie release date",
    "type":"integer",
    "regex":"\\d{4}"
    },
    "cover":{
    "description": "movie photo",
    "type":"string",
    "presentationMode":"image"
    },
    "trailer":{
    "description": "movie trailer",
    "type":"string",
    "presentationMode":"video"
    }
    },
```

```
"required": ["name","year","cover","trailer"],
"references": [{
"model": "Category",
"relation": "1-M",
"label": "LabelCategory"
}
,{
"model": "Director",
"relation": "1-M",
"label": "LabelDirector"
}]
}
```

No inicio é definido o nome, descrição e tipo do modelo, posteriormente seguem-se as propriedades onde cada uma contém os seguintes atributos:

- **type** (obrigatório): Define o tipo de variável cujos valores da propriedade serão (ex.: String, Number, etc.);
- **description**: Uma breve descrição do que a propriedade representa no domínio do problema;
- **unique** (opcional): Se é um valor único;
- **regex**: Define um padrão que o valor de uma propreidade terá que cumprir.
- **presentationMode**: Define o tipo de objeto a ser apresentado sendo por default text mas podendo assumir outros valores como image ou video.

As propriedades que serão obrigatórias de introduzir na aplicação são definidas no campo *required* do modelo.

Nas referências, é possível determinar o **nome** do modelo a que este estará interligado, a tipo de **relação** dessa ligação, o nome da propriedade (**label**) a apresentar na aplicação e, opcionalmente, se esta é necessária ser introduzida na criação de um objecto do modelo principal.

Por exemplo:

```
{
"model": "Category",
"relation": "1-M",
"label": "LabelCategory"
}
```

Isto define que o modelo *Category* estará relacionado a um ou mais *Movie* e o campo a apresentar será o **LabelCategory** do *Category*.

De forma a resolver o problema apresentado foi criados os seguintes modelos para o domínio:

- Actor;
- Category;
- Director;
- Movie;

Geração de código

BackOffice de criação de modelos

Foi criada uma aplicação *BackOffice* para facilitar a criação dos modelos e fazer as alterações necessárias à aplicação de forma a personalizar da melhor forma possível.

Transformação

Transformação: Conceito de converter, mantendo os seus valores, um modelo em outro, ou em texto (código). As transformações que serão referidas neste manual serão transformações modelo-para-texto, que, como já indicado, consistirão na conversão dos modelos criados em código *JavaScript*.

Para realizar as transformações modelo-para-texto, foram utilizados templates *Mustache*, que permite combinar um objecto *JavaScript* com um ficheiro com etiquetas embutidas, onde serão escritos os valores do objecto.

Por exemplo, se considerarmos o seguinte ficheiro *template.mustache*:

```
<body>{{{menu}}}</body>
```

E o seguinte código *JavaScript*:

```
var params = {
    menu: "<p>Menu da App</p>"
};

var template = fs.readFileSync("./template.mustache").toString();

var output = mustache.render(template, params);
```

O *output* será:

```
<p>Menu da App</p>
```

De seguida segue um exemplo mais concreto de um template *mustache*, o mustache apresentado a seguir é uma parte da página de listar de um modelo na aplicação:

```
<body>

{{{menu}}}<br/>

<button  type="button"  class="btn btn-success"  style="margin-right:10px;"  onc

<input  type="text"  id="inputSearch"  onkeyup="search(this)"  placeholder="Sear

<table  id="table">

<tr>
```

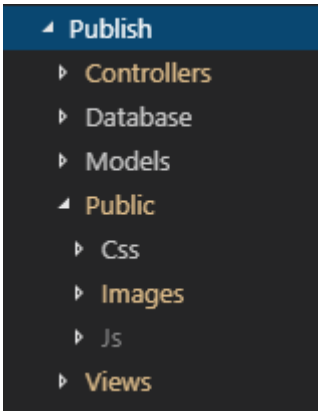
```
{{#columns}}
<th>{{.}}</th>
{{/columns}}
</tr>
{{#rows}}
<tr>
{{#properties}}
<td>
{{value}}
</td>
{{/properties}}
{{#actions}}
<td>
<a href="{{link}}"  title="{{tooltip}}"  {{#events}}{{name}}="javascript:{{funct
{{#image}}
<img src="{{src}}"  alt="{{alt}}"  />
{{/image}}
</td>
{{/actions}}

</tr>
{{/rows}}
</table>
```

```
</body>
```

Uma etiqueta em *Mustache* seguida de um "#" significa que o texto lá dentro terá um de dois tipos de comportamentos: caso o objecto seja um *array*, o texto será iterado para cada elemento, caso seja uma variável de valor lógico, apenas uma vez. É de notar, também, que o símbolo que indica o *else* perante um valor lógico é o "^".

Criação de um novo servidor

O primeiro passo na geração da nova aplicação será a criação da estrutura. Através das bibliotecas **mkdirp**, **fs** e **del**, é criada uma estrutura de pastas apresentada pela seguinte imagem:



Para a criação do servidor da nova aplicação, é utilizado, também, templates *Mustache* para criar o seguinte ficheiro:

```
var express = require("express");

var mustache = require('mustache');

var mustacheExpress = require('mustache-express');

var app = express();

var fs = require('fs');

const bodyParser = require("body-parser");

const api = require('./Controllers/api.js');
```

```
const backOffice = require('./Controllers/backoffice.js');

const frontOffice = require('./Controllers/frontoffice.js');


app.engine('mustache', mustacheExpress());

app.set('view engine', 'mustache');

app.set('views', __dirname + '/Views');


app.use(bodyParser.json());

app.use(bodyParser.urlencoded({extended: false}));

app.use('/api', api);

app.use('/backoffice', backOffice);

app.use('/frontoffice', frontOffice);


app.get('/', function(req, res){

var template = fs.readFileSync("./Publish/Views/template.mustache").toString();


var template_config = {

title: 'Backoffice',

menu: getMenu()

};


var output = mustache.render(template, template_config);

res.send(output);
```

```
});

function getMenu (){
var menu = fs.readFileSync("./Publish/Views/menu.mustache").toString();

schemas = [];
var config = JSON.parse(fs.readFileSync("./Server/config.json"));

config.schemas.forEach(model => {
schemas.push(model);
});

var menu_config = {
schemas: schemas
}

return mustache.render(menu, menu_config);
}

app.use(express.static(__dirname + '/Public'));

var server = app.listen({{port}},function () {
var host = server.address().address === "::" ? "localhost" :
server.address().address
```

```
var port = server.address().port

console.log("Published Server running on http://%s:%s", host, port)

});
```

Através da biblioteca **nodemon** é possível iniciar o novo servidor e este ser recarregado sem ter necessidade de mandar a aplicação abaixo.

```
function  startPublish(){
        server =  nodemon({ script:  './Publish/index.js' }).on('start', functic
        console.log('server started');
}).on('crash', function () {
        console.log('server crashed...');
});

console.log(server);
}
```

No final são copiados para cada devida pasta os ficheiros estáticos necessários por parte da aplicação criada tal como ficheiros *CSS* ou *JS*. Só depois é que se passa para a geração de classes e tabelas na base de dados, baseadas nos modelos.

Geração de classes e base de dados

Base de dados:

Primeiramente, é necessário criar a base de dados. É chamado o *database-generator*.

```
function  generateDatabase(schemas, db_name, dummyData) {

db  =  new  sqlite3.Database("./Publish/Database/"  +  db_name);



schemas.forEach(schema  => {

createTable(schema);

});
```

```
setTimeout(function () {
schemas.forEach(schema  => {
generateRelationships(schema);
});
}, 2500);

setTimeout(function () {
if(dummyData)
createDummyData();
}, 3000);

setTimeout(function () {
db.close();
}, 4000);
}
```

Este indica que, para cada modelo, será criada uma tabela, utilizando cada propriedade para construir uma coluna da tabela dependendo do *type*:

```
function  createTable(schema) {
var  props  =  schema.properties;
var  keys  =  Object.keys(props);
var  str_collumns_boy  =  "";
```

```
var query_template = fs
.readFileSync("./Models/Database/create-table.mustache")
.toString();

keys.forEach(key => {
str_collumns_boy += createCollumn(schema, key);
});

str_collumns_boy = str_collumns_boy.slice(0, -1);

var config = {
table_name: schema.title,
primary_key: schema.title.toLowerCase() + "_id",
collumns_body: str_collumns_boy
};
db.run(mustache.render(query_template, config));
}
```

```
CREATE TABLE IF NOT EXISTS {{table_name}} (
    {{primary_key}} integer PRIMARY KEY AUTOINCREMENT,
    {{{collumns_body}}}
)
```

E, de seguida, criadas as chaves estrangeiras necessárias, baseadas nas referências do modelo:

```
function generateRelationships(schema) {
var references = schema.references;
var config_json = JSON.parse(fs.readFileSync("./Server/config.json"));
```

```
var  query_template;
if (references  !==  void  0) {
references.forEach(reference  => {
var  config  = {
child_table:  schema.title,
parent_table:  reference.model,
child_collumn:  schema.title.toLowerCase(),
parent_collumn:  reference.model.toLowerCase()
};
switch (reference.relation) {
case  "1-1":
query_template  =  fs.readFileSync("./Models/Database/fk_11.mustache").toString(
db.run(mustache.render(query_template, config));
setTimeout(function () {
query_template  =  fs.readFileSync("./Models/Database/fk_11_secondary.mustache")
db.run(mustache.render(query_template, config));
}, 1000);
break;
case  "1-M":
query_template  =  fs.readFileSync("./Models/Database/fk_1M.mustache").toString(
db.run(mustache.render(query_template, config));
break;
```

```
case "M-M":
if (
mmrelations.indexOf(
config.child_table + "_" + config.parent_table
) == -1 &&
mmrelations.indexOf(
config.parent_table + "_" + config.child_table
) == -1
) {
if (config_json.databases === void 0)
config_json.databases = [];

var exists = false;

config_json.databases.some(table => {
if (table.tableName.toLowerCase() == config.child_table.toLowerCase() + "_" +
table.tableName.toLowerCase() == config.parent_table.toLowerCase() + "_" + c
exists = true;

return exists == true;
});

if (!exists) {
config_json.databases.push({
```

```
tableName:  config.child_table  +  "_"  +  config.parent_table

});



fs.writeFileSync("./Server/config.json", JSON.stringify(config_json));

}



query_template  =  fs

.readFileSync("./Models/Database/fk_MM.mustache")

.toString();

db.run(mustache.render(query_template, config));

mmrelations.push(config.child_table  +  "_"  +  config.parent_table);

}

break;

}

});

}

}
```

Quando a aplicação não possui dados também é executado a função *createDummyData* na qual irá popular a base de dados criada.

```
function  createDummyData(){

console.log("Generating dummy data...");

db.exec("INSERT INTO Actor (name, birthyear) VALUES ('TomasEdi', 1999),('KongoPc

db.exec("INSERT INTO Category (name) VALUES ('Horror'),('Comedy'),('Action'),('D
```

```
db.exec("INSERT INTO Director (name, birthyear) VALUES ('Agustini', 1980),('Piri

db.exec("INSERT INTO Movie (name, year, cover, trailer, category_id, director_ic

db.exec("INSERT INTO Actor_Movie (actor_id,movie_id) VALUES (1,1),(1,2),(1,3),(3

}
```

Exemplo de um ficheiro de alteração da base de dados, neste caso, de uma chave estrangeira M-M, onde é criada uma nova tabela para contemplar a referida relação:

```
CREATE TABLE IF NOT EXISTS {{{child_table}}}_{{{parent_table}}}(
    {{{child_collumn}}}_id INTEGER NOT NULL,
    {{{parent_collumn}}}_id INTEGER NOT NULL,
    PRIMARY KEY ({{{child_collumn}}}_id , {{{parent_collumn}}}_id),
    FOREIGN KEY ({{{child_collumn}}}_id) REFERENCES {{{child_table}}}({{{child_c
    FOREIGN KEY ({{{parent_collumn}}}_id) REFERENCES {{{parent_table}}}({{{parer
)
```

Geração de classes:

Na geração de classes são definidas as propriedades que cada uma irá ter e é construido passo a passo a composição de cada atributo. Também é feito o mapeamento destes atributos com as respectivas colunas da base de dados, além do constructor e respectivas referencias na classe.

Foi abordado o problema da mesma forma que nos laboratórios, sendo apenas apresentado a realização das referências na classe:

```
function generateReferences(schema) {

var references = schema.references;

var str = "";

var model = "";

if (references !== void 0) {

references.forEach(reference => {
```

```
model  =  reference.model.toLowerCase();


switch (reference.relation) {
case  "1-1":
str  +=
'\n\t\tObject.defineProperty(this, "'  +
model  +
'_id",{\n\t\t\t enumerable: false, writable: true \n\t\t});';
break;


case  "0-M":
str  +=
'\n\t\tObject.defineProperty(this, "'  +
model  +
'_id",{\n\t\t\t enumerable: false, writable: true \n\t\t });';
break;


case  "1-M":
str  +=
'\n\t\tObject.defineProperty(this, "'  +
model  +
'_id",{\n\t\t\t enumerable: false, writable: true \n\t\t });';
break;
```

```
case  "M-M":
str  +=
'\n\t\tObject.defineProperty(this, "'  +
model  +
'_ids",{\n\t\t\t enumerable: false, writable: true \n\t\t, value:[] });';
break;
}
});
}
return  str;
}
```

É de notar, nesta abordagem, que os valores de uma referência muitos-para-muitos, é feito através de um *array* desse valor.

API RESTful

A geração da API é bastante simples. Para cada modelo, é gerado uma função para cada um dos verbos RESTful (GET, POST, PUT e DELETE), que, por sua vez, chamam funções já criadas nas classes que fazem operações na base de dados.

Segue um exemplo dessas funções:

```
Brand.all = function (callback) {
    database.all("SELECT * FROM Brand" , Brand, callback);
}

Brand.get = function (id, callback) {
    database.get("SELECT * FROM Brand WHERE brand_id = ? ",[id] ,Brand, callback
```

```
}

Brand.delete = function (id, callback) {
    database.get("DELETE FROM Brand WHERE brand_id = ? ",[id] ,Brand, callback);
}

Brand.prototype.save = function (callback) {
    if(this.id) {//UPDATE
        database.run("UPDATE Brand SET name = ? WHERE brand_id = ?",
        [this.name

            ,this.id
        ]
        , callback);

    } else {//INSERT
        database.run("INSERT INTO Brand(name ) VALUES(?)" ,
         [this.name

         ]);
    }
}
```

Para solucionar o problema de obter informação sobre a junção de duas tabelas, para relações M-M cria-se funções GET para cada um dos modelos da relação:

```
{{#nmRelations}}
router.get("/{{{modelA}}}/:model/:id", function(req, res) {
    {{{modelA}}}.many(req.params.model, req.params.id, function(rows) {
        res.json(rows);
    });
});

router.get("/{{{modelB}}}/:model/:id", function(req, res) {
    {{{modelB}}}.many(req.params.model, req.params.id, function(rows) {
        res.json(rows);
    });
});
{{/nmRelations}}
```

Exemplo da função *many* utilizada:

```
Sale.many = function (model, id, callback) {
    var nmTableName = database.getNMTableName(Sale, model.constructor.name);
```

```
    database.where(`SELECT Sale.* FROM Sale INNER JOIN ${nmTableName} ON ${nmTab
    Sale.id WHERE ${nmTableName}.${model.toLowerCase()}_id = ?`, [id], Sale, cal
}
```

Após isso, a única coisa que falta é apresentar páginas *web* ao utilizador.

Renderização de páginas web

De forma a ser mais eficiente e não criar de forma exponencial várias páginas *web* para cada modelo, todas as páginas da aplicação são criadas do lado do servidor e renderizadas no cliente através da função **renderPage**.

Para cada modelo, no *BackOffice* eram criados os verbos necessários para renderizar cada página das CRUDs do mesmo. Para o *FrontOffice*, apenas se teve de definir a apresentação da página inicial. No template do mesmo, criou-se uma função genérica que apresentava um "top modelos" baseado num critério, também ele genérico e fornecido por parâmetros, como tal:

```
{{{frontoffice.model}}}.top("{{{frontoffice.property}}}","{{{frontoffice.order}}

            {{{frontoffice.model}}}.topWithMMCount("{{{frontoffice.MMModel}}}","
                console.log(results);
                console.log(rows);
                renderPage(req, res , 'welcome.mustache', {
                    welcome: 'Welcome to '+ config.website_name,
                    motto: config.homepage.motto,
                    rows: {
                            topProductModel: rows.length > 0 ? rows.map(obj => {
                                return {
                                    id: obj.id,
                                    name: obj.name,
                                    price : obj.price,
                                    description: obj.description,
                                    model:"{{{frontoffice.model}}}"
                                }
                            }) : false,
                            topSalesModel: results.length > 0 ? results.map(obj
                                return {
                                    id: obj.id,
                                    name: obj.name,
                                    price : obj.price,
                                    description: obj.description,
```

```
                                    count: obj.count,
                                    model:"{{{frontoffice.model}}}"
                                }
                            }) : false
                        }
                    })
                });
            })
```

O template da página inicial em si seria configurada da seguinte forma:

```
<div class="container">
    <h2> Top Sales </h2>
      <div class="row" style="margin-top:2%;">
        {{#rows}}
          {{#topSalesModel}}
            <div class="col-md-4">
              <h2>{{name}}</h2>
              <h3>{{price}}€</h3>
              <p>{{description}}</p>
              <p>Vendas: {{count}}</p>
              <p><a class="btn btn-secondary" href="/backoffice/{{{model}}}/Deta
            </div>
          {{/topSalesModel}}
          {{^topSalesModel}}
            <div class="col-md-4">
              <p>No products to display here yet!</p>
            </div>
          {{/topSalesModel}}
        {{/rows}}
      </div>
    <h2>Top Products </h2>
        <div class="row" style="margin-top:2%;">
          {{#rows}}
            {{#topProductModel}}
              <div class="col-md-4">
                <h2>{{name}}</h2>
                <h3>{{price}}€</h3>
                <p>{{description}}</p>
                <p><a class="btn btn-secondary" href="/backoffice/{{{model}}}/De
              </div>
            {{/topProductModel}}
            {{^topProductModel}}
              <div class="col-md-4">
```

```
                <p>No products to display here yet!</p>
              </div>
            {{/topProductModel}}
          {{/rows}}
        </div>
  </div>
```

É de notar, também, que foi criada uma página geral que contém o *layout* geral da aplicação, onde depois é só preciso "injectar" o corpo, através de um objecto por sua vez renderizado por outro *template*. Segue parte do mesmo:

```
<html>
        <head>
                <title>{{title}}</title>
        </head>
        <body>
                {{{menu}}}
                <br  />
                {{{page}}}
        </body>
</html>
```

Problemas relevantes encontrados

Para resolver problemas derivados de tabelas M-M e da confusão que, em termos de programação, podiam surgir visto que é aleatório qual a ordem dos nomes dos modelos associados que a tabela terá, foi criado um atributo no ficheiro de configuração denominado de *databases* que guarda o nome das tabelas M-M, após isso, foi criada a seguinte função no ficheiro de SQLite para se aceder a esta tabela de forma rápida:

```
        getNMTableName: function getNMTableName(model1, model2)
        {
          var tableName = "";

          var config_json =  JSON.parse(fs.readFileSync("./Server/config.json"))

          config_json.databases.some(table => {
            var models = table.tableName.split("_");

            if((models[0] == model1 && models[1] == model2) || (models[1] == moc
              tableName = table.tableName;
```

```
            return tableName != "";
          });

          return tableName;
        }
```

Limitações do Projecto

Devido à tecnologia utilizada para as transformações modelo-para-texto, o *Mustache* torna-se um pouco limitado devido a ser *logicless*, não permitindo assim a habitual utilização dos *statements if*, tornando o código por vezes complexo.
Atualmente também existem problemas com o eliminar/criar a estrutura de pastas visto estas por vezes estarem ainda em *cache* e não conseguirem ser apagadas devido estarem abertas noutro lugar.