UNIVERSIDADE da MADEIRA

ESCOLA SUPERIOR DE TECNOLOGIAS E GESTÃO

Ano Letivo 2020/2021

Curso Técnico Superior Profissional em: Tecnologias e Programação de Sistemas de Informação

2 º **Ano**/1 º **Semestre**

Unidade Curricular: Sistemas Gestores de Bases de Dados II

Docente: Magno Andrade **Época:** Normal

BREVE EXPLICAÇÃO DO EXPRESS JS E MONGOOSE

Express JS

É uma *framework* de NodeJS para a criação de aplicações web e móveis do lado do servidor.

Instalação

Para a utilizar, é preciso instalá-la no projecto pretendido, com o seguinte comando:

npm install express --save

Rotas

São os "caminhos" para uma determinada função ou funcionalidade implementada quando executado um URL.

Exemplo de Rotas

```
#index.js

...

const router = express.Router() // 1

router.get('/', (req, res) => {  // 2
  res.send('This is the homepage!')
})

router.post('/contact', (req, res) => {  // 3
  res.send('This is the contact page with a POST request')
})

app.use('/', router)  // 4

...
```

1 – Uma instância do *Express Router* é criada. Esta instância é referenciada por uma variável de nome *router*. Esta variável pode ser utilizada sempre que se pretenda criar uma nova rota.

2 – Uma nova rota é definida do tipo GET, para a raiz da aplicação. Está associada à instância da classe *Express Router*.

3 – Uma nova rota é definida do tipo POST, para a página (caminho) *contact* da aplicação. Está associada à instância da classe *Express Router*.

4 – É a associação para a utilização de um *middleware,* para lidar com as rotas criadas. Isto permite que para cada pedido (HTTP) realizado à aplicação para o caminho "/", o uso das rotas criadas.

Nota: se fosse utilizado para o ponto 4, o seguinte código:

```
app.use('/user', router)
```

As rotas criadas só iriam ser utilizadas se no prefixo do URL, estivesse por exemplo:

1. /user/profile
2. /user/profile/edit
3. /user/dashboard/article/view

Rotas – Métodos

Os métodos das rotas são derivados dos pedidos HTTPs que existem e associados à instância do *Express Router*.

Alguns métodos de rotas que o ExpressJS suporta e que correspondem a métodos HTTP são:

- GET

- POST
- PUT
- DELETE

Rotas – Parâmetros

São usados para obter valores que estão em certas posições de um URL. São chamados de segmentos URL. Os valores obtidos são disponibilizados através do objecto ***req.params***, utilizando o nome do parâmetro escolhido no caminho, como chave para obter esse valor.

Exemplo de Parâmetros

Existindo uma rota com o seguinte URL -> **/users/:userId/articles/:articleId**

O URL usado para realizar o pedido, por exemplo:
http://localhost:3000/users/19/articles/104

O objecto ***req.params*** seria composto por: **{ "userId": "19", "articleId": "104" }**

Mongoose

É um *Object Data Model ("ODM"),* para efectuar a interacção com a base de dados, esta interacção é feita com métodos já implementados e sem a utilização da linguagem nativa do MongoDB.

Instalação

Para a utilizar, é preciso instalá-la no projecto pretendido, com o seguinte comando:

npm install mongoose --save

Conexão

É necessário realizar a conexão com o servidor do MongoDB, por exemplo, é usado o seguinte código:

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/test');
```

Para detectar se a conexão foi realizada com sucesso:

```
var db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));

db.once('open', function() {

  // we're connected!

});
```

Esquemas

Tudo no Mongoose, começa como um esquema.

Cada esquema mapeia uma colecção do MongoDB e define a forma/estrutura dos documentos dessa colecção.

Exemplo:

```
var schema = new mongoose.Schema({ name: 'string', size: 'string' });
```

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

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

Uma propriedade é definida pelo seu *SchemaType* (ex: *String, Number*)

Alguns exemplos de *SchemaTypes*:

- **String**
- **Number**
- **Date**
- **Boolean**
- **ObjectId**
- **Array**
- **Decimal128**

Modelos

Os esquemas criados tem de ser compilados num modelo.

Os modelos são compilados através de uma definição de um esquema.

Exemplo:

```
var Tank = mongoose.model('Tank', schema);
```

CRUD

Uma *query* **mongoose** pode ser executada de duas formas, com ou sem *callback*.

Criar documentos (*Create*)

Uma instância de um modelo é um documento.

Exemplo:

```
var Tank = mongoose.model('Tank', yourSchema);

var small = new Tank({ size: 'small' });
small.save(function (err) {
  if (err) return handleError(err);
  // saved!
});

// or

Tank.create({ size: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
});

// or, for inserting large batches of documents
Tank.insertMany([{ size: 'small' }], function(err) {

});
```

Pesquisa de documentos (*Read*) com *callback*:

A pesquisa é efectuada e os documentos são retornados no interior da função, ao mesmo tempo.

Exemplo:

```
var Person = mongoose.model('Person', yourSchema);

// find each person with a last name matching 'Ghost', selecting the `name` and `occupation` fields
Person.findOne({ 'name.last': 'Ghost' }, 'name occupation', function (err, person) {
  if (err) return handleError(err);
  // Prints "Space Ghost is a talk show host".
  console.log('%s %s is a %s.', person.name.first, person.name.last,
    person.occupation);
});
```

Pesquisa de documentos (*Read*) sem *callback*:

A pesquisa é definida mas só é efectuada mais tarde e os documentos são retornados nessa fase.

Exemplo:

```
// find each person with a last name matching 'Ghost'
var query = Person.findOne({ 'name.last': 'Ghost' });

// selecting the `name` and `occupation` fields
query.select('name occupation');

// execute the query at a later time
query.exec(function (err, person) {
  if (err) return handleError(err);
  // Prints "Space Ghost is a talk show host."
  console.log('%s %s is a %s.', person.name.first, person.name.last,
    person.occupation);
});
```

Actualização de documentos (*Update*) com retorno:

É procurado o documento por id, este documento é retornado para dentro da função, os campos são alterados, e por fim é guardado as alterações do documento.

Exemplo:

```
Tank.findById(id, function (err, tank) {
  if (err) return handleError(err);

  tank.size = 'large';
  tank.save(function (err, updatedTank) {
    if (err) return handleError(err);
    res.send(updatedTank);
  });
});
```

Actualização de documentos (*Update*) sem retorno:

É procurado o documento pelo critério de pesquisa (neste caso, com o campo ***size*** que corresponde a ***large***), o campo ***name*** é alterado para ***T-90***.

Neste caso o documento não é retornado para a aplicação.

Exemplo:

```
Tank.updateOne({ size: 'large' }, { name: 'T-90' }, function(err, res) {
  // Updated at most one doc, `res.modifiedCount` contains the number
  // of docs that MongoDB updated
});
```

Remoção de documentos (*Delete*) sem retorno:

```
Tank.deleteOne({ size: 'large' }, function (err) {
  if (err) return handleError(err);
  // deleted at most one tank document
});
```

Alguns comandos para realizar operações CRUD:

- Model.deleteMany()
- Model.deleteOne()
- Model.find()
- Model.findById()
- Model.findByIdAndDelete()
- Model.findByIdAndRemove()
- Model.findByIdAndUpdate()
- Model.findOne()
- Model.findOneAndDelete()
- Model.findOneAndRemove()
- Model.findOneAndUpdate()
- Model.replaceOne()
- Model.updateMany()
- Model.updateOne()

Subdocumentos

Subdocumentos são documentos embebidos em outros documentos. No Mongoose significa que é possível criar esquemas dentro (“aninhados”) de outros esquemas. Existem dois tipos de subdocumentos:

- *Array* de subdocumentos.
- Um só subdocumento.

Exemplo:

```
var childSchema = new Schema({ name: 'string' });

var parentSchema = new Schema({
  // Array of subdocuments
  children: [childSchema],
  // Single nested subdocuments. Caveat: single nested subdocs only work
  // in mongoose >= 4.2.0
  child: childSchema
});
```

No Mongoose, os subdocumentos são similares aos documentos mas com uma diferença, estes não são guardados individualmente. É necessário guardar o documento "pai" para que sejam gravadas as modificações dos subdocumentos.

Exemplo:

```
var Parent = mongoose.model('Parent', parentSchema);
var parent = new Parent({ children: [{ name: 'Matt' }, { name: 'Sarah' }] })
parent.children[0].name = 'Matthew';

// `parent.children[0].save()` is a no-op, it triggers middleware but
// does **not** actually save the subdocument. You need to save the parent
// doc.
parent.save(callback);
```

Procurar subdocumento por _id

Cada subdocumento tem um campo *_id* por defeito. Existe um método ***id*** para encontrar um subdocumento através do seu campo *_id.*

Exemplo:

```
var doc = parent.children.id(_id);
```

Procurar subdocumentos por um atributo ou critério específico

É possível procurar subdocumentos por determinados atributos ou por critérios de pesquisa personalizados.

Exemplo:

Tendo em conta o seguinte *array* de documentos definido como *parents*.

```
var parents = [
    { name: "John Smith",
    children: [
        { name: "Peter", age: 2 }, { name: "Margaret", age: 20 }
    ]},
    { name: "Another Smith",
    children: [
        { name: "Martha", age: 10 }, { name: "John", age: 22 }
    ]}
];
```

É pretendido obter todos os subdocumentos ***children,*** em que campo ***age*** é maior ou igual que 18, retornando apenas os subdocumentos que satisfazem esse critério de pesquisa.

```
db.parents.find(
    {'children.age': {$gte: 18}},
    {children:{$elemMatch:{age: {$gte: 18}}}})
```

É obtido o seguinte *output*:

```
{ "_id" : ..., "children" : [ { "name" : "Margaret", "age" : 20 } ] }
{ "_id" : ..., "children" : [ { "name" : "John", "age" : 22 } ] }
```

Nota: se é necessário apenas os subdocumentos (sem estes estarem agrupados no documento "pai"), existe a possibilidade de utilizar a *framework* ***aggregate,*** para obter como *output* apenas os subdocumentos como documentos individuais.

```
> db.parents.aggregate({
    $match: {'children.age': {$gte: 18}}
}, {
    $unwind: '$children'
}, {
    $match: {'children.age': {$gte: 18}}
}, {
    $project: {
        name: '$children.name',
        age:'$children.age'
    }
})
{
    "result" : [
        {
            "_id" : ObjectId("51a7bf04dacca8ba98434eb5"),
            "name" : "Margaret",
            "age" : 20
        },
        {
            "_id" : ObjectId("51a7bf04dacca8ba98434eb6"),
            "name" : "John",
            "age" : 22
        }
    ],
    "ok" : 1
}
```

Adicionar subdocumento a um *array* de subdocumentos

Os *arrays* no Mongoose possuem já métodos para a manipulação destes *arrays*. Existem por exemplo, os seguintes métodos: ***push***, ***unshift***.

Exemplo:

```
var Parent = mongoose.model('Parent');
var parent = new Parent;

// create a comment
parent.children.push({ name: 'Liesl' });
var subdoc = parent.children[0];
console.log(subdoc) // { _id: '501d86090d371bab2c0341c5', name: 'Liesl' }
subdoc.isNew; // true

parent.save(function (err) {
  if (err) return handleError(err)
  console.log('Success!');
});
```

Exemplo:

```
var newdoc = parent.children.create({ name: 'Aaron' });
```

Remover um subdocumento

Cada subdocumento tem o seu próprio método ***remove***. Num *array* de subdocumentos é o equivalente a executar o método ***.pull()*** num determinado subdocumento. Num só subdocumento, o método ***remove()*** é o equivalente a colocar o conteúdo do campo que detém o subdocumento a ***null***.

Exemplo:

```
// Equivalent to `parent.children.pull(_id)`
parent.children.id(_id).remove();
// Equivalent to `parent.child = null`
parent.child.remove();
parent.save(function (err) {
  if (err) return handleError(err);
  console.log('the subdocs were removed');
});
```

Referências

Existem também os casos em que os modelos estão normalizados, a definição do esquema nestes casos deve ser realizada como apresentado no exemplo seguinte:

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

var personSchema = Schema({
  _id: Schema.Types.ObjectId,
  name: String,
  age: Number,
  stories: [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

var storySchema = Schema({
  author: { type: Schema.Types.ObjectId, ref: 'Person' },
  title: String
});

var Story = mongoose.model('Story', storySchema);
var Person = mongoose.model('Person', personSchema);
```

São criados dois modelos. Um modelo ***Person*** que tem um campo ***stories*** que é um *array* de *ObjectIds*. A opção ***ref*** é o que fornece ao *Mongoose*, o modelo a ser utilizado na ***Population***, neste caso o modelo ***Story***. Todos os ***_ids*** são ***_ids*** dos documentos do modelo ***Story***.

Por exemplo, os tipos *Number*, String *são* válidos como ***refs***. No entanto, devemos utilizar o *ObjectId*.

Guardar as referências

Para guardar os ***_ids*** das referências devemos utilizar o código apresentado na imagem abaixo:

```
var author = new Person({
  _id: new mongoose.Types.ObjectId(),
  name: 'Ian Fleming',
  age: 50
});

author.save(function (err) {
  if (err) return handleError(err);

  var story1 = new Story({
    title: 'Casino Royale',
    author: author._id    // assign the _id from the person
  });

  story1.save(function (err) {
    if (err) return handleError(err);
    // thats it!
  });
});
```

Pesquisa com retorno dos dados da colecção de referência:

Também chamado de ***Population***, é o processo de repor nos caminhos especificados do documento por outros documentos de outras colecções (equivalente à etapa ***$lookup*** da *framework* ***aggregate***).

Exemplo:

```
Story.
  findOne({ title: 'Casino Royale' }).
  populate('author').
  exec(function (err, story) {
    if (err) return handleError(err);
    console.log('The author is %s', story.author.name);
    // prints "The author is Ian Fleming"
  });
```