

Conectando ao banco de dados

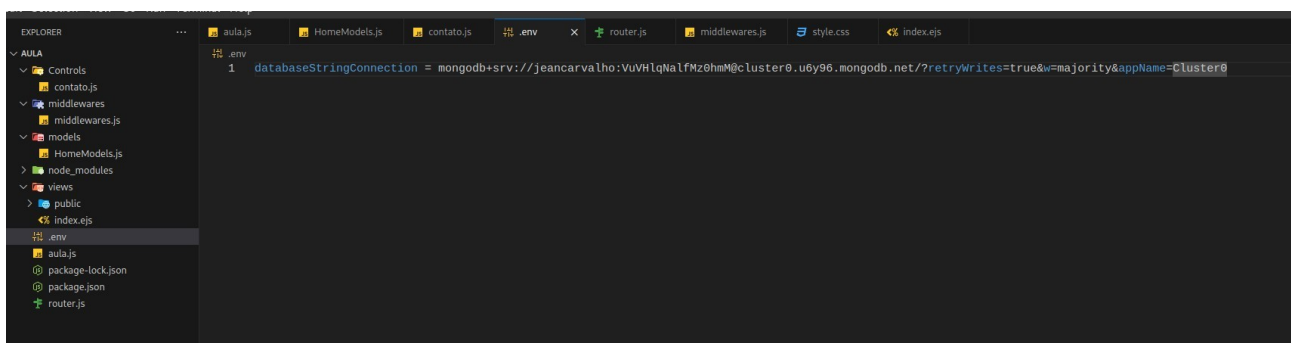
vamos importar duas bibliotecas importantes: **dotenv** e **mongoose**.

dotenv: serve para você importar chaves para variavel global **process.env** do **node.js**. Assim, deixando a sua senha para se conectar ao db fora dos scripts principais através de um arquivo **.env**.

mongoose: serve para você se conectar ao seu banco de dados mongodb.

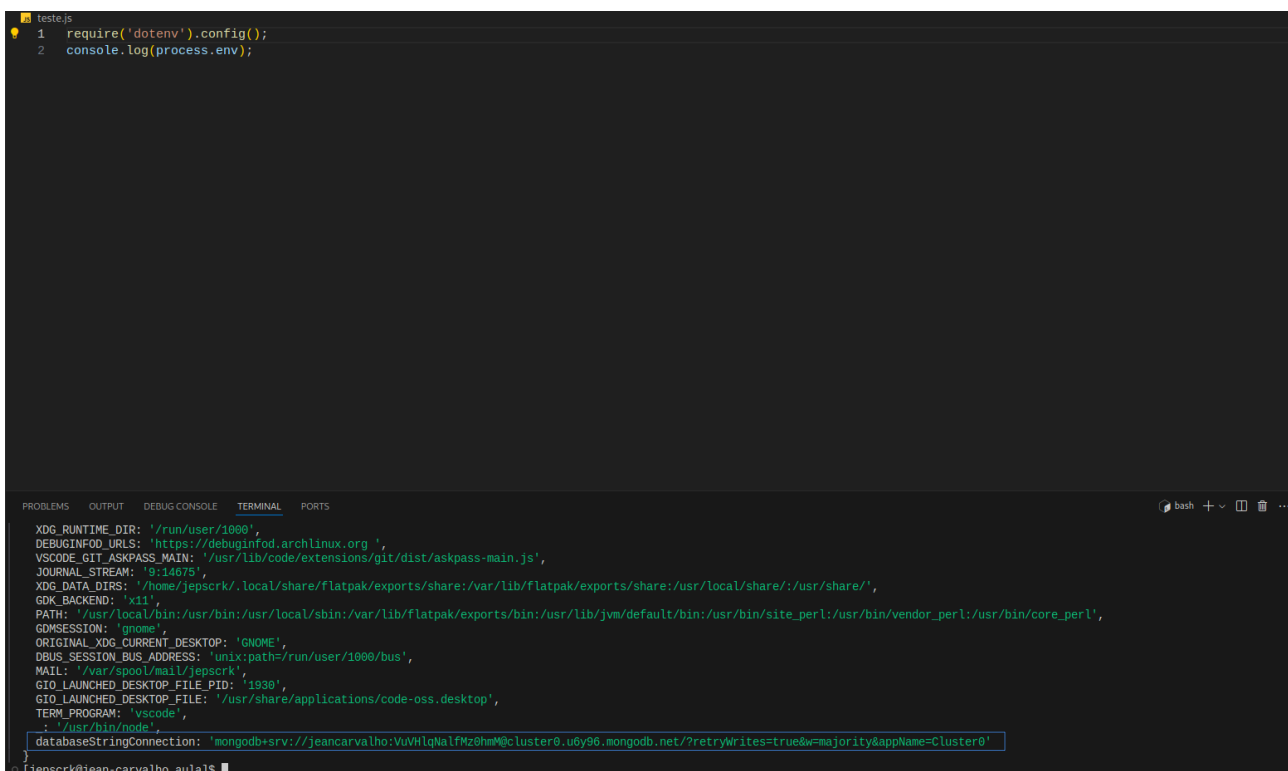
Primeiro vamos criar um arquivo **.env** para colocar a chave do db

Obs.: o arquivo tem que está presente na raiz do projeto



```
require('dotenv').config();
```

vamos importar o modulo dessa maneira e adicionar o **.config()** - para que o dotenv envie as chaves e valores para variavel global do node.js. Ex:



Sabendo disso, agora vamos importar a chave do db para que nós possamos nos conectar ao mongodb:

```
js aula.js > ...
1  require('dotenv').config();
2  const express = require('express');
3  const router = require('./router.js')
4  const path = require('path');
5  const app = express();
6  const middleware = require('./middlewares/middlewares.js');
7  const mongoos = require('mongoose');
8
9  mongoos.connect(process.env.databaseStringConnection).then(() => {
10 |     console.log('Database: Connection made');
11 |     app.emit('db connection'); //emitindo sinal
12 | }).catch(e => console.log('Database: Error'));
13
14  app.use(express.urlencoded({extended: true}));
15
16  app.use(express.static(path.resolve(__dirname, 'views', 'public')))
17
18  app.set('views', path.resolve(__dirname, 'views'));
19  app.set('view engine', 'ejs');
20
21  app.use(middleware);
22  app.use(router);
23
24  app.use((req, res) =>{
25 |     res.status(404).send('<h1>404 not found</h1>');
26 | });
27  app.on('db connection', () =>{ //recebendo sinal
28 |     app.listen(3001, () =>{
29 |         console.log('Acessar: http://localhost:3001');
30 |         console.log('Servidor executando');
31 |     });
32 | });
33
```

Vamos levantar um assunto importante: Quando iniciamos o db e o nosso servidor express, geralmente o express inicia primeiro e o db por último. Caso o db esteja com problema e só express funcionar, pode acontecer alguns problemas de conexões do db para o usuário e não queremos isso. Vamos fazer uma conexão assíncronica para o db primeiro está on para que só depois o express iniciar também.

Mas como vamos fazemos isso? Simples! Com **.emit()** e **.on()** o **.emit()** emite um sinal “db connection” e o **.on()** recebe esse sinal para poder iniciar o server

Obs.: pode ser qualquer sinal

```
Database: Connection made
Acessar: http://localhost:3001
Servidor executando
```

Agora vamos criar nossos dados e armazenar no db

Primeiro passo: Iremos criar uma pasta com o nome models e criar um arquivo HomeModels.js (*nome opcional do arquivo .js*).

Lá dentro iremos criar nosso schema – serve para definir a estrutura dos seus dados

```
const mongoose = require('mongoose');
const home = new mongoose.Schema({
  userName: {type: String, required: true},
  Local: String
});
const homeModel = mongoose.model('home', home);
module.exports = homeModel;
```

```
const home = new mongoose.Schema({
  userName: {type: String, required: true},
  Local: String
});
```

Inicia uma nova instância e assim você insere as chaves e a estrutura de como os dados devem ser preenchidos.

O que é um Model no Mongoose?

O **Model** é a representação da **coleção no MongoDB**. Ele é criado com base no Schema e permite interagir com o banco de dados.

Exemplo de Model:

```
const homeModel = mongoose.model('home', home);
```

Aqui, estamos criando um **modelo chamado homeModel**, que:

- Está associado à coleção **"home"** no MongoDB.
- Segue a estrutura definida no **homeSchema**.

Com esse Model, podemos:

- Criar documentos (`new homeModel({...})`).
- Salvar no banco (`.save()`).
- Consultar dados (`.find()`).
- Atualizar dados (`.updateOne()`, `.updateMany()`).
- Remover dados (`.deleteOne()`, `.deleteMany()`).

Nesse caso, lembre-se que o nome 'home' é opcional, você pode colocar 'casa', por exemplo. Você irá ver o resultado disso lá na plataforma do mongodb onde ele definiu o nome do modelo com o nome que você colocou.

Vimos como schematizar os dados para inserir no db, mas como faz para inserir os dados agora?

Iremos usar uma função .create()

```
const db = require('../models/HomeModels');

db.create({
  userName: 'Jean',
  Local: 'guamá'
})
```

Existe uma função .find para você encontrar os dados

```
exports.home = (req, res) =>{
  res.render('index');
  db.find({userName: 'Jean'})
    .then(dados => console.log(dados))
    .catch(e => console.log(e));
}
```

se você deixar o argumento de find vazio, ele irá retornar todos

lembrando que as funções .create() e .find() retornam uma promessa

Vamos armazenar os dados no cookie

Quando o usuário fizer autenticação uma vez, iremos fazer que ele sempre fique logado. Pra isso, iremos importar duas bibliotecas: **express-session** e **connect-mongo**

Express-session: é um middleware que serve para você armazenar as sessões na memória;

Connect-mongo: serve para você armazenar as sessões no banco de dados mongodb;

```
const session = require('express-session');  
const connect = require('connect-mongo');
```

Agora vamos iniciar:

```
const refreshSession = session({
  secret: 'h1uo238cajkh',
  store: connect.create({
    mongoUrl: process.env.databaseStringConnection,
    ttl: 1000 * 60 * 60 * 24 * 14,
    autoRemove: 'native'
  }),
  saveUninitialized: false,
  resave: false,
  cookie: {
    maxAge: 1000 * 60 * 60 * 24 * 14,
    httpOnly: true,
    secure: false
  }
})
app.use(refreshSession);
```

iremos iniciar o express-session como session para executar mais tarde

secret: é o que será que você irá armazenar

store: é o local do armazenamento, e é ai que iremos usar o connect-mongo

que está como connect, ai iremos usar o paramentro .create(que é um objeto)

mongoUrl: é a url do link do seu mongodb

ttl (time to live): o tempo de vida que a sessão irá ficar mantida

1000 – milsegundos, 60 – segundos, 60 – hora, 24 um dia, 14 dias

então fica em 24horas depois de 7 dias irá expirar

autoRemove: É como ele irá remover as sessões ativas, existem 3 tipos de argumentos:

Valor	Descrição
'native'	O próprio MongoDB remove as sessões expiradas automaticamente usando TTL Indexes .
'interval'	O connect-mongo cria um processo em segundo plano que remove sessões expiradas em um intervalo de tempo definido .
false	Não remove sessões automaticamente (você precisará gerenciá-las manualmente).

exemplo do uso 'interval':

```
autoRemove: 'interval', // Remove sessões expiradas a cada intervalo de tempo
autoRemoveInterval: 10 // O intervalo de remoção será a cada 10 minutos
```

ou seja, a cada 10 min ele irá verificar quem já expirou pra remover

Secure: Faz que os cookies sejam enviados apenas em HTTPS se for **true**, como estamos em desenvolvimento, então é **false**, já que ele irá mandar tanto em http, quanto https, mas em versão final, mude para **true**

saveUninitialized(salvar não inicializado): ele irá salvar mesmo se os dados estiverem vazios, ou seja, se **saveUninitialized** fosse **true**, assim que um usuário acessasse o site, uma sessão vazia seria criada e salva, mesmo que nada fosse armazenado nela.

Resave: evita de salvar as sessões sem estarem modificadas
controla se a sessão deve ser salva no armazenamento mesmo que ela não tenha sido modificada.

cookie: define como a sessão irá ser armazenada e gerenciada no navegador do usuário

maxAge: é o tempo que irá se manter até expirar

httpOnly: garante que o cookie seja acessado só pelo servidor(back-end) e que não passe pelo javascript no lado do cliente, protege contra ataque xss ou seja, O cookie **não pode** ser acessado via `document.cookie` no navegador. Apenas o servidor pode ler e modificar

e para você acessar os dados armazenados, quando você usa o express-session você ativa um recurso 'session' e assim, você pode visualizar e armazenar os dados

```
exports.home = (req, res) =>{  
  req.session.usuario = {nome: 'joao', logado: true}  
}
```

já armazenamos, agora iremos ver:

```
exports.home = (req, res) =>{  
  res.send(`olá, ${req.session.usuario.nome}`);  
  console.log(req.session);  
}
```

deu até pra brincar com os nomes

```
15 console.log(req.session);  
16 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Restarting 'aula.js'  
Database: Connection made  
Acessar: http://localhost:3000  
Servidor executando  
middleware test connection  
Session {  
  cookie: {  
    path: '/',  
    _expires: 2025-02-27T14:39:00.603Z,  
    originalMaxAge: 1209600000,  
    httpOnly: true  
  },  
  usuario: { nome: 'jean', logado: true },  
  flash: {  
    success: [ 'Operação realizada com sucesso!' ],  
    error: [ 'Ocorreu um erro!' ],  
    info: [ 'Informações importantes para você!' ]  
  }  
}
```


Página de login

Usaremos o nosso conhecimento para fazer autenticação, criar conta, fazer login, criptografia e manter a sessão.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>register</title>
  <link rel="stylesheet" href="/assets/css/style-register.css">
</head>
<body>
  <main>
    <section class="content-register">
      <header>
        <h1>Sing up</h1>
      </header>
      <div class="box-create">
        <form action="/admin" method="POST" class="form-class">
          <input type="hidden" name="_csrf", value="{%= csrfToken %}">
          <input type="text" name="user[username]" id="user-id-registrer" class="user-box-input" maxlength="30" minlength="3"
            placeholder="create your id" required autocomplete="off">
          <input type="text" name="user[email]" id="user-email" class="user-box-input" maxlength="30"
            placeholder="typing email" required autocomplete="off">
          <input type="password" name="user[pass]" id="user-pass-registrer" class="user-box-input" maxlenght="50"
            minlength="6" placeholder="create your password" required>
          <input type="password" name="user[passcheck]" id="user-pass-registrer-check" class="user-box-input"
            maxlenght="30" minlength="6" placeholder="create your password" required>
          <button type="submit" class="btn-form">Create account</button>
        </form>
      </div>
      <div>
        <p>You have accout?<span><a href="/login"> Login</a></span></p>
      </div>
      <div>
        <%- include("message") %>
      </div>
    </section>
  </main>
</body>
</html>
```

O include('message') é onde iremos incluir mensagens de erro ou sucesso, quando o usuário errar algo como senhas diferentes, e-mail inválido e mensagens de sucesso quando a conta for criada.

Lá no arquivo models, iremos criar um schema e um model, lá iremos trabalhar na validação dos valores inseridos, na 'limpeza' para que não tenha espaços e iremos garantir que será valores seja do tipo string (não confiamos em usuários)

```
server.js  middleware.js  router.js  homeModels.js X  form.js
models > homeModels.js > Login > cleanUp
1 const mongo = require('mongoose');
2 const validator = require('validator');
3 const login = new mongo.Schema({
4   username: {type: String, require: true},
5   email: {type: String, require: true},
6   pass: {type: String, require: true}
7 });
8 const model = mongo.model('login', login);
9
10 class Login{
11   constructor(body){
12     this.body = body;
13     this.error = [];
14     this.user = null;
15   }
16   async register(){
17     this.valida();
18     if(this.error.length > 0) return;
19     try{
20       this.user = await model.create(this.body.user); // estamos acessando o user de fora
21     }catch(e){
22       console.log(e);
23     }
24   }
25   valida(){
26     this.cleanUp();
27     if(!validator.isEmail(this.body.user.email)) this.error.push('invalid email');
28     if(this.body.user.pass.trim() !== this.body.user.passcheck.trim()) this.error.push('different passwords');
29     if(this.body.user.pass.length < 3 || this.body.user.pass.length > 50) this.error.push('Password must be between 3 and 50 characters');
30   }
31   cleanUp(){
32     for(const key in this.body.user){
33       if(typeof this.body.user[key] !== 'string'){
34         this.body.user[key] = '';
35       }
36     }
37     this.user = {
38       username: this.body.user.username,
39       email: this.body.user.email,
40       pass: this.body.user.pass,
41       passcheck: this.body.user.passcheck
42     }
43   }
44 }
45
```

```
const login = new mongo.Schema({
  username: {type: String, required: true},
  email: {type: String, required: true},
  pass: {type: String, required: true},
  sessionId: {type: String, default: null}
});
```

É required e não require

Dentro da classe, podemos ver 3 funções que mencionamos, e 3 propriedades no objeto(body, erros, user). irei explicar por detalhes que cada uma serve:

this.body = irá receber o body do formulário;

this.erros = um array onde iremos receber os erros que e alertar o usuário de um determinado erro

this.user = objeto do usuário com os dados após a validação, para salvar

1. na classe login iremos usar fazer uma constructor para recebe esse parametros

2. na função register que irá registrar o usuário, porém, antes os dados serão passados por mais duas funções, da validação dos dados que chama a função de limpeza para garantir que seja do tipo string

! Na função cleanUp se os valores não forem string, ele deixa os valores vazios

! A cada erro que a função valida achar, ele coloca no array

Passando pelas essas duas funções, iremos pro register que é uma função async, se o array de erro for maior, ele não irá cadastrar o user no db, se tiver tudo certo, iremos colocar um await na variavel user para ele salvar e acessar fora do db.

Vamos para o arquivo controls

```
13 exports.create = (req, res) => {
14   console.log(req.session);
15   res.render('create.ejs', {
16     csrfToken: res.locals.csrfToken,
17     errors: req.flash('errors'),
18     success: req.flash('success')
19   });
20 }
21 exports.admin = async(req, res) => {
22   try{
23     console.log(req.session);
24     const login = new Login(req.body);
25     await login.register();
26     if(login.error.length > 0){
27       req.flash('errors', login.error);
28       req.session.save(function(){
29         return res.redirect('create');
30       });
31       return
32     }
33     req.flash('success', 'Account create!');
34     req.session.save(function(){
35       res.redirect('create');
36     })
37   }catch(e){
38     console.log(e);
39   }
40 }
41
```

.create() é função que rederiza a pagina de criar conta

iremos passar duas váriaveis locais para receber as mensagens de erros usando a biblioteca flash. Calma, já iremos trabalhar com elas

.admin() é a requisição via POST

```
router.post('/admin', page.admin);
```

Lembrando que fizemos a importação daquela classe Login

```
const Login = require('../models/homeModels');
```

Por essa ser uma classe construtora, iremos iniciar uma instância, e fazendo e chamando logo o login. Como lá fizemos ele async, nós iremos coloca-lo em await envolvido em um try e catch. Agora vamos fazer uma pequena validação, se conter algum erro, iremos salvar a sessão. A função `req.session.save()` **garante que as alterações na sessão sejam salvas antes de redirecionar o usuário**. No seu código, você está armazenando mensagens de erro ou sucesso na sessão (`req.flash()`). Como o redirecionamento acontece logo depois, pode haver casos em que a sessão **não seja salva a tempo**, o que faria com que as mensagens não aparecessem corretamente na página seguinte.

Conclusão: A sessão é salva para garantir que as mensagens de erro/sucesso sejam persistidas antes do redirecionamento, evitando problemas onde os dados se perdem no meio do processo.

Ou seja, para parecer as mensagens daquela sessão que o usuário fez login, temos que salvar para parecer na próxima requisição.

Esse `res.redirect('create')` é que ele está voltando do admin para o create para requisição anterior, poderíamos também usar o `'back'`, mas ele não se enquadra nesse exemplo porque retorna para uma outra página(você lembra)

Se for sucesso, mesma coisa, iremos passar pro flash mensagem de sucesso, salvar a sessão e voltar para o create com mensagem que a conta foi criada.

OBS: poderíamos usar um middleware para armazenar o token, as mensagens de erro e sucesso assim:

```

routes.js  loginController.js  LoginModel.js  login.ejs  index.ejs  messages.ejs  middleware.js
> middleware.js > middlewareGlobal > exports.middlewareGlobal
1  exports.middlewareGlobal = (req, res, next) => {
2    res.locals.errors = req.flash('errors');
3    res.locals.errors = req.flash('errors');
4    next();
5  };
6
7  exports.outroMiddleware = (req, res, next) => {
8    next();
9  };
10
11 exports.checkCsrfError = (err, req, res, next) => {
12   if(err) {
13     return res.render('404');
14   }
15
16   next();
17 };
18
19 exports.csrfMiddleware = (req, res, next) => {
20   res.locals.csrfToken = req.csrfToken();
21   next();
22 };

```

```

exports.create = (req, res) => {
  res.render('create.ejs'); // As variáveis já estão disponíveis no template!
};

```

Isso elimina a necessidade de passar manualmente as variáveis para `res.render()` em todas as rotas.

```

view > message.ejs > ? > ? > ? > ?
1  <% if(errors && errors.length > 0) { %>
2    <div class="message_erros_alert">
3      <% errors.forEach(e => { %>
4        <p><%= e %></p>
5      <% }); %>
6    </div>
7  <% }; %>
8  <% if(success){ %>
9    <div class="message_success_alert">
10     <p><%= success %></p>
11   </div>
12 <% } %>

```

Vamos falar brevemente de flash() - ela serve para você armazenar mensagens temporária

Você baixa a biblioteca connect-flash, ela precisa do express-session, pois é nela que o flash fica armazenado

```
const session = require('express-session');

app.use(helmet());
app.use(refresh);
app.use(csrf());
app.use(csrfMiddleware);
app.use(flash());
app.use(router);
```

you inicia o flash depois do session iniciado

ele é bem simples para o uso, ele é um objeto, quando você inicia a função, você passa a chave e o valor

```
req.flash('errors', 'deu erro');
```

pronto, agora para você iniciar, acessar o valor, você chama pelo valor:

```
req.flash('errors')
```

Mas atenção: numa requisição é salva as mensagens, na outra é exibida e na outra é excluída automaticamente

Mas se tiver um e-mail já cadastrado ?

Vamos importar uma função do model chamado **.findOne()**, adicionar em register e criar uma função que irá verificar se o e-mail tá cadastrado

```
async register() {
  this.valida();
  if (this.error.length > 0) return;
  await this.Exists();
}
```

Como iremos acessar o db, então terá que ser async e await

```
async Exists() {
  const user = await model.findOne({email: this.body.user.email});
  if (user) this.error.push('Email já está registrado');
}
}
```


Criptografia

quando criamos a conta, a senha do user chegou no db assim:

```
1  _id: ObjectId('67bcafa053cfde17ce187ad4')
2  username: "admin"
3  email: "admin@gmail.com"
+  pass: "jean123"
5  v . a
```

Com a senha exposta, e não é nada seguro ou recomendado, então usaremos o `bcryptjs`

```
const bcrypt = require('bcryptjs');
```

```
async register(){
  this.valida();
  if(this.error.length > 0) return;
  await this.Exists();

  if(this.error.length > 0) return;

  const salt = bcrypt.genSaltSync(10);
  this.body.user.pass = bcrypt.hashSync(this.body.user.pass, salt);
  this.user = await model.create(this.body.user); // estamos acessando o user de fora
}

valida(){
  this.cleanUp();
  if(!validator.isEmail(this.body.user.email)) this.error.push('invalid email');
  if(this.body.user.pass.trim() !== this.body.user.passcheck.trim()) this.error.push('different passwords');
  if(this.body.user.pass.length < 3 || this.body.user.pass.length > 50) this.error.push('Password must be between 3 and 50 characters');
}

cleanUp(){
  for(const key in this.body.user){
    if(typeof this.body.user[key] !== 'string'){
      this.body.user[key] = '';
    }
  }
  this.user = {
    username: this.body.user.username,
    email: this.body.user.email,
    pass: this.body.user.pass,
    passcheck: this.body.user.passcheck
  }
}
```

Antes dele criar a conta, ele irá criar uma hash na senha, no qual passará por duas camadas:

Gera um **salt** que usa a função **bcrypt.genSaltSync(10)** (valor aleatório) que será usado para "**salgar**" a senha antes de ser criptografada. Isso impede que senhas idênticas tenham hashes iguais. O número passado para `genSaltSync(10)` define a complexidade do salt. Quanto maior, mais seguro, mas também mais demorado. Usa o salt gerado para criar um **hash** da senha. Isso significa que, mesmo que duas pessoas tenham a mesma senha, os hashes serão diferentes, tornando ataques como *rainbow table* ineficazes.

E o **hashSync(pass, salt);**

que irá criar um hash de acordo com os caracteres passados junto com salt

Fazendo login

Temos nossa conta criada, agora vamos fazer o login. Iremos criar uma função método POST para autenticação

```
exports.authentic = async(req, res) => {
  try{
    const login = new Login(req.body);
    await login.autenticar();
    if(login.error.length > 0){
      req.flash('errors', login.error);
      req.session.save(function(){
        return res.redirect('login');
      })
      return;
    }
    req.flash('success', 'login realizado')
    req.session.user = login.user
    console.log(req.session);
    req.session.save(function(){
      res.redirect(['login'])
    })
  }catch(e){
    console.log(e);
  }
}
```


Iremos criar uma função autenticar() que irá verificar se os dados estão cadastrados e se estão certos

```
async autenticar(){
  this.user = await model.findOne({email: this.body.user.email});
  if(!this.user || !bcrypt.compareSync(this.body.user.pass, this.user.pass)){
    this.error.push('Email ou senha inválida');
    this.user = null
    return;
  }
}
```

Lembra que a nossa senha está com uma hash, como backend vai entender que é a nossa senha ? Usando o **bcrypt.compareSync(senha, senha_com_hash)** como ele compara os hash passados?

Como `bcrypt.compareSync()` verifica a senha?

Quando você usa `bcrypt.compareSync(senhaDigitada, hashArmazenado)`, ele **não precisa retirar o salt manualmente**. O próprio hash armazenado **já contém o salt embutido**. O processo funciona assim:

- 1 O usuário digita a senha no login.
- 2 O `bcrypt.compareSync()` pega o hash armazenado no banco.
- 3 Ele extrai o salt de dentro do hash armazenado.
- 4 Usa esse salt para gerar um novo hash da senha digitada.
- 5 Compara o novo hash com o hash armazenado no banco.
- 6 Se os hashes forem iguais, a senha está correta! 

De forma bem simples, ele irá verificar se o e-mail passado está logado, ele irá retornar todos os dados

```
user: {
  _id: '67bcafa053cfde17ce187ad4',
  username: 'admin',
  email: 'admin@gmail.com',
  pass: '$2b$10$38qjnJthn0phnL5G05W3F.Rp.rwaV5EywGeFsXUogI9a0CG7jH0RG',
  __v: 0
}
```

já retornado, o que irá acontecer agora:

O `req.session.user` está armazenando o `login.user` que são os dados que o banco de dados retornou pra gente

```
this.user = await model.findOne({email: this.body.user.email});
```

feito isso, o que acontece, com uma pagina de dashboard especifica para os usuários autenticados, ele só irá carregar se os dados do user estiver em `req.session.user`

```
exports.dashboard = (req, res) => {
  if(req.session.user){
    res.render('dashboard', {
      user: req.session.user.username
    });
    return
  }
  res.redirect('login');
```

Se não tiver, ele irá redirecionar para o login, em login, é mesma coisa. Iremos adicionar uma condição para que não aparece a página de login

```
exports.login = (req, res) => {  
  if(req.session.user){  
    res.redirect('dashboard');  
    return  
  }  
  res.render('login.ejs', {  
    csrfToken: res.locals.csrfToken,  
    success: req.flash('success'),  
    errors: req.flash('errors')  
  });  
}
```

pagina de user:

```
dashboard.ejs x login.ejs create.ejs message.ejs style-login.css style-  
view > dashboard.ejs > html > body > h2 > ?  
1 <!DOCTYPE html>  
2 <html lang="en">  
3 <head>  
4   <meta charset="UTF-8">  
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">  
6   <title>Dashboard</title>  
7 </head>  
8 <body>  
9   <h2>Olá, <%= user %> %></h2>  
10 </body>  
11 </html>
```