



Web Data Viz

Funcionamento do Web Data Viz

TÓPICOS



Back-end x Front-end



Conceitos a serem entendidos



Status HTTP



Estrutura do diretório do Web Data Viz



Mão na Massa

Front-end

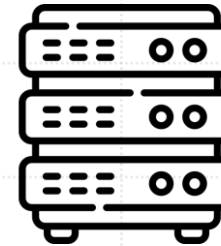


O cliente tem acesso

Responsável por **estilizar a tela e tratar os dados que vem e vão** ao back-end.

O usuário comum tem acesso a parte do código fonte
(HTML / CSS / JS)

Back-end



O cliente não tem acesso

Responsável por **transacionar os dados** vindos do front-end **até o banco de dados e retornar dados já existente.**

O usuário comum nunca terá acesso a nenhuma parte do código fonte.

Objetivos do back-end

- Dar vida ao sistema realizando os CRUDs (Inserção, leitura, atualização e remoção).
- Servir o front-end para interação com o usuário.



Conceitos a serem entendidos antes...

O nome correto de rota é endpoint!

Um endpoint é a “rota” para a qual a requisição deverá ser enviada.

O **status HTTP** é uma resposta numérica indicando o resultado de uma requisição. Ele é composto por um código de três dígitos, onde o primeiro dígito define a classe do status.

Existem os verbos HTTP, mas estes serão explicados durante os próximos slides.

npm install

Instala as bibliotecas necessárias para a aplicação.

npm start

Inicia a aplicação na máquina local

1XX
Informativo

2XX
Sucesso

3XX
Redirecionamento

4XX
Erro de Cliente

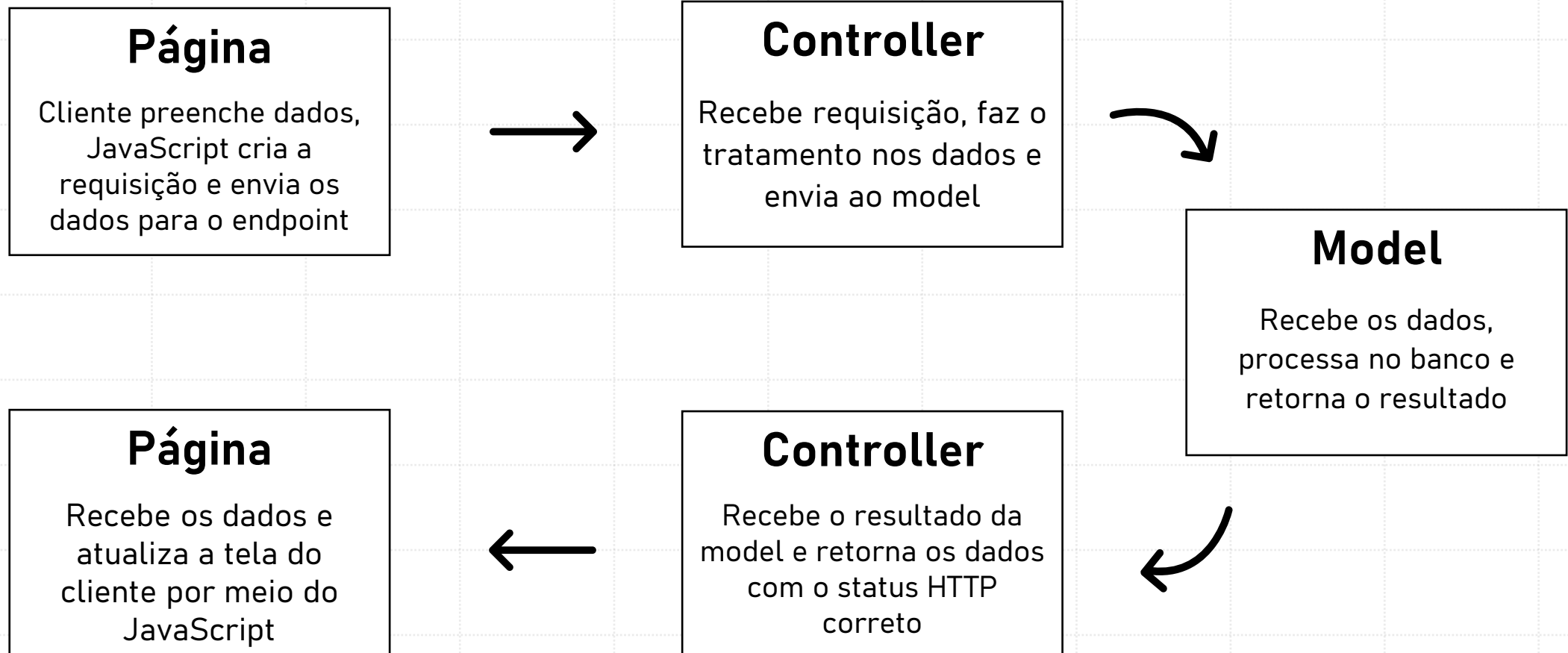
5XX
Outros Erros

STATUS HTTP MAIS USADOS

Código	Status	Descrição
200	OK	Indica que deu tudo certo.
201	CREATED	Indica que um recurso foi criado com sucesso.
204	NO CONTENT	Indica que uma consulta não encontrou resultados.
400	BAD REQUEST	Indica que alguma coisa nos parâmetros e/ou corpo da requisição está errada.
401	UNAUTHORIZED	Indica que era necessário alguma credencial de acesso.
403	FORBIDDEN	Indica que não é possível liberar o acesso.
404	NOT FOUND	Indica que o recurso solicitado não existe.
409	CONFLICT	Indica que o recurso já foi criado.
500	INTERNAL SERVER ERROR	Indica que ocorreu algum erro não tratado no serviço.

Se quiser descobrir mais sobre os códigos... <http.cat>









Fluxo comum do Web Data Viz



Conhecendo o nosso diretório

Antes, precisamos entender como funcionam os nossos diretórios e arquivos.

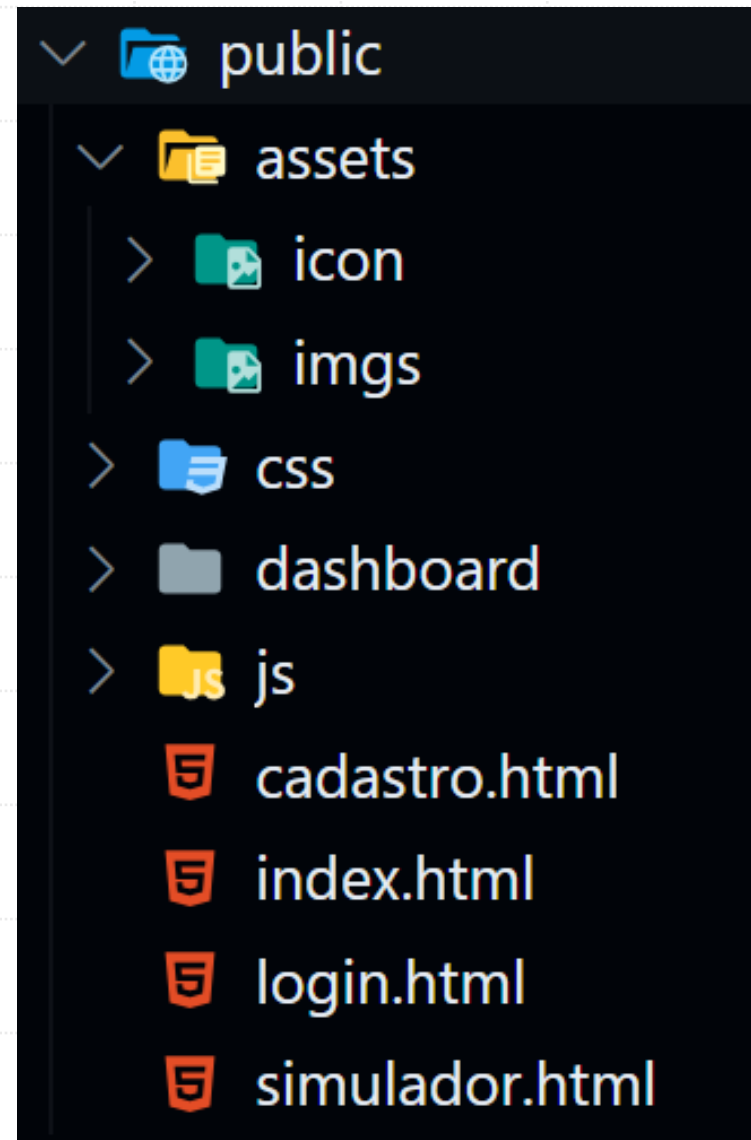
web-data-viz

- >  public → Onde fica localizado o front-end
- ▼  src → Onde fica os arquivos do back-end
 - >  controllers → Camada responsável por tratar e devolver a requisição
 - >  database → Arquivos de configuração do banco
 - >  models → Camada responsável por se comunicar com o banco
 - >  routes → Camada responsável por definir as “rotas” até as controllers
-  app.js → Arquivo de configuração da aplicação
-  package.json → Catálogo de bibliotecas a serem baixadas pelo Node

O que é o diretório public?

Nele alocaremos os **arquivos HTML, CSS e JS** que desenvolvemos para o nosso grupo de PI e para o nosso Projeto Individual.

Ou seja, o nosso **front-end**!





Vamos juntos fazer o login?

IMPORTANTE**

Para não se perder, use como base o código já existente do Web Data Viz e modifique ele para suprir as necessidades do site de vocês.

<https://github.com/BandTec/web-data-viz.git>

Formulário de login



Olá de volta!

E-mail:

Senha:

Entrar

<http://localhost:3333/login.html>

```
<div class="formulario">
  <div class="campo">
    <span>E-mail:</span>
    <input id="email_input" type="text" placeholder="Login">
  </div>
  <div class="campo">
    <span>Senha:</span>
    <input id="senha_input" type="password" placeholder="*****">
  </div>
  <button class="botao" onclick="entrar()">Entrar</button>
</div>
```

login.html

Ao iniciar a aplicação e acessar a **tela de login**, observe a presença de **um botão** que, ao ser clicado, **invoca a função “entrar”** dentro do código JavaScript

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

A função “entrar” atribui os valores do inputs a variáveis e os valida.
Não esqueça de fazer as validações da sua regra de negócio!

```
function entrar() {  
  aguardar();  
  
  var emailVar = email_input.value;  
  var senhaVar = senha_input.value;  
  
  if (emailVar == "" || senhaVar == "") {  
    cardErro.style.display = "block"  
    mensagem_erro.innerHTML = "(Mensagem de erro para todos os campos em branco)";  
    finalizarAguardar();  
  
    return false;  
  } else {  
    setInterval(sumirMensagem, 5000)  
  }  
}
```

login.html

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Estamos utilizando a função “fetch” para enviar a requisição.

Precisamos colocar o endpoint (rota) para onde essa requisição deverá chegar e o corpo da requisição.

```
fetch("/usuarios/autenticar", {  
  method: "POST",  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify({  
    emailServer: emailVar,  
    senhaServer: senhaVar  
  })  
})
```

login.html

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado


Controller


Mais tarde explicaremos o porquê do nome emailServer e senhaServer**

O que eu preciso saber sobre esse tal de fetch?

O fetch é uma função do JavaScript que é usada para fazer requisições de rede, geralmente para obter ou enviar dados para um servidor web, que é a sua aplicação Node correndo com npm start!

Ele precisa de um **endpoint** (rota) e de um **corpo de requisição** como parâmetros...

 Endpoint

 Corpo da Requisição

```
fetch("/usuarios/autenticar", {  
  method: "POST",  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify({  
    emailServer: emailVar,  
    senhaServer: senhaVar  
  })  
})
```





Preciso me preocupar com ele?

Ele não cairá na prova!

Não precisa ficar assustado, mas é sempre bom entender o conceito do que estamos mexendo...

Voltaremos a falar sobre envio requisições na matéria de **Programação Web**
Aguarde cenas dos próximos capítulos...

O que colocamos no corpo da requisição?

Deixamos explicito por meio da chave "headers" de que a requisição está levando dados em JSON.

E colocamos os dados catalogados dos inputs dentro da chave body utilizando o formato JSON.

```
fetch("/usuarios/autenticar", {  
  method: "POST",  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify({  
    emailServer: emailVar,  
    senhaServer: senhaVar  
  })  
})
```

login.html

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller



PAUSA NO FLUXO

Por que eu estou colocando `"/usuarios/autenticar"`?

Por que eu estou colocando `"method": "POST"`?

Explicando o endpoint...

No arquivo **app.js**, configuramos que todas as requisições que vierem com o caminho inicial “/usuarios” devem procurar o arquivo usuarios.js dentro do diretório “routes”.

```
var usuarioRouter = require("./src/routes/usuarios");  
  
app.use("/usuarios", usuarioRouter);
```

app.js

Explicando o endpoint...

Olha só o que o arquivo usuarios.js dentro de “routes” tem?

Um bloco de códigos que indica o **final do endpoint**, qual é o **tipo de requisição** e aponta **para onde a requisição deve ir**.

```
router.post("/autenticar", function (req, res) {  
  usuarioController.autenticar(req, res);  
});
```

routes/usuarios.js

Certo, mas o que é POST?

POST faz parte de uma lista de **métodos HTTP** que são usados em requisições.

Os métodos HTTP servem para deixar explícito qual é o objetivo de cada requisição.

Método	Descrição
GET	Indica que irá listar ou buscar dados.
POST	Indica que irá enviar dados.
PUT	Indica que irá atualizar dados.
DELETE	Indica que irá deletar dados.

Estes são os 4 métodos mais usados nas requisições, não lembra um pouco o **CRUD** que aprendemos em Banco de Dados?



VOLTANDO PARA O FLUXO

Agora já sabemos como está configurado os endpoints e quais são os melhores métodos HTTP para cada requisição.

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

A função “autenticar” está dentro de usuarioController.js

Foi para ela que pedimos que a “route” enviasse as requisições que chegassem com método POST no endpoint “usuarios/autenticar”

```
function autenticar(req, res) {  
  var email = req.body.emailServer;  
  var senha = req.body.senhaServer;  
  
  if (email == undefined) {  
    res.status(400).send("Seu email está undefined!");  
  } else if (senha == undefined) {  
    res.status(400).send("Sua senha está indefinida!");  
  } else {
```

usuarioController.js

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Observe que temos como primeiro parâmetro a requisição (req) e a resposta (res).

A requisição é aquele objeto enviamos com as chaves, contendo "method", "header" e "body".

```
function autenticar(req, res) {  
  var email = req.body.emailServer;  
  var senha = req.body.senhaServer;  
  
  if (email == undefined) {  
    res.status(400).send("Seu email está undefined!");  
  } else if (senha == undefined) {  
    res.status(400).send("Sua senha está indefinida!");  
  } else {
```

usuarioController.js

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Por meio do req conseguimos acessar o body, a chave onde colocamos o e-mail e a senha do usuário.

Então atribuímos os valores que vieram na requisição dentro de algumas variáveis.

```
function autenticar(req, res) {  
  var email = req.body.emailServer;  
  var senha = req.body.senhaServer;  
  
  if (email == undefined) {  
    res.status(400).send("Seu email está undefined!");  
  } else if (senha == undefined) {  
    res.status(400).send("Sua senha está indefinida!");  
  } else {
```

usuarioController.js

Agora você entendeu por que nomeamos as chaves com o sufixo “Server”?

```
body: JSON.stringify({  
  emailServer: emailVar,  
  senhaServer: senhaVar  
})
```

login.html

Inserimos o valor que está alocado na variável **emailVar** para dentro da chave **emailServer**.

```
function autenticar(req, res) {  
  var email = req.body.emailServer;  
  var senha = req.body.senhaServer;
```

usuarioController.js

E conseguimos recuperar o email por meio da chave **emailServer** que havia recebido o valor vindo do input.

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Por meio do objeto req conseguimos acessar o body, a chave onde colocamos o e-mail e a senha do usuário.

Validamos se os dados chegaram corretamente, caso contrário, enviamos uma resposta (res) com **status 400 (BAD REQUEST)**, avisando que os dados chegaram de forma incorreta.

```
function autenticar(req, res) {  
  var email = req.body.emailServer;  
  var senha = req.body.senhaServer;  
  
  if (email == undefined) {  
    res.status(400).send("Seu email está undefined!");  
  } else if (senha == undefined) {  
    res.status(400).send("Sua senha está indefinida!");  
  } else {
```

usuarioController.js

Dica: Dê uma olhada na tabela de status HTTP

Quando validado, chamamos o model, passando o e-mail e a senha do usuário.

```
if (email == undefined) {  
    res.status(400).send("Seu email está undefined!");  
} else if (senha == undefined) {  
    res.status(400).send("Sua senha está indefinida!");  
} else {  
  
    usuarioModel.autenticar(email, senha)  
}
```

usuarioController.js

FLUXO

Página

Cliente insere seus dados, nós validamos e mandamos para a controller

Controller

Recebe a requisição, faz o tratamento e envia ao model

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

FLUXO

Controller

Recebe a requisição,
faz o tratamento e
envia ao model

Model

Recebe os dados,
processa no banco e
retorna o resultado

Controller

Recebe o resultado da
model e retorna os
dados com o status
HTTP correto

Página

Dentro da model executamos a query necessária.

No caso, seria um **SELECT**, mas dependendo da necessidade, pode ser qualquer comando executável no banco de dados.

```
function autenticar(email, senha) {  
  var instrucao = `  
    SELECT id, nome, email, fk_empresa as empresaId  
    FROM usuario  
    WHERE email = '${email}' AND senha = '${senha}';`;  
  
  return database.executar(instrucao);  
}
```

usuarioModel.js

FLUXO

Assim que é retornado a resposta da model, utilizamos o **then** (então) para validar se o resultado retornou apenas 1 linha.

Ele funciona assim como o banco, o **SELECT** que foi feito **deve retornar apenas 1 linha** de resultado. Qual é o sentido de ter dois usuários com o mesmo e-mail e senha?

```
usuarioModel.autenticar(email, senha)
    .then(
        function (resultadoAutenticar) {
            if (resultadoAutenticar.length == 1) {
```

usuarioController.js

Funciona parecido com o MySQL Workbench, ele deve retornar apenas uma linha.

```
SELECT id, nome, email, fk_empresa AS idEmpresa
FROM usuario
WHERE email = 'leonardo.vasconcelos@email.com' AND senha = '123456';
```

id	nome	email	idEmpresa
1	Leonardo Vasconcelos	leonardo.vasconcelos@email.com	1

1 row(s) returned

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

FLUXO

Model

Recebe os dados, processa no banco e retorna o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

Na regra de negócio do Aquatech, assim que o usuário realiza o login, pegamos também quais são os aquários de sua empresa.

Em seguida, a model da entidade "aquário" é acionada, utilizando o resultado da consulta SELECT do usuário para enviar à empresa à qual ele pertence.

```
.then(  
  function (resultadoAutenticar) {  
    if (resultadoAutenticar.length == 1) {  
  
      aquarioModel.buscarAquariosPorEmpresa(resultadoAutenticar[0].empresaId)    }  
  })
```

usuarioController.js

FLUXO

Observe que referenciamos o **“resultadoAutenticar” na posição 0** como parâmetro. Isso ocorre porque o resultado retornado pelo SELECT é um array de ocorrências do banco de dados, semelhante às linhas que o MySQL Workbench retorna.

Então como há apenas um usuário e desejamos pegar a empresa dele, acessamos a posição 0 e utilizamos o **campo “empresaid”** para passar como parâmetro de qual empresa deve ser buscada.

```
.then(  
  function (resultadoAutenticar) {  
    if (resultadoAutenticar.length == 1) {  
  
      aquarioModel.buscarAquariosPorEmpresa(resultadoAutenticar[0].empresaId)
```

usuarioController.js

Model

Recebe os dados,
processa no banco e
retorna o resultado

Controller

Recebe o resultado da
model e retorna os
dados com o status
HTTP correto

Página

Recebe os dados e
atualiza a tela do
cliente por meio do
JavaScript

Como fica o retorno da model visualmente?

O SELECT que fizemos trouxe apenas um resultado:

```
[
  {
    "id": 1,
    "nome": "Leonardo Vasconcelos",
    "email": "leonardo.vasconcelos@email.com",
    "empresaId": 1
  }
]
```

Vetor do JavaScript

Mas e se fizéssemos um SELECT que retornasse vários resultados?

```
[
  {
    "id": 1,
    "nome": "Leonardo Vasconcelos",
    "email": "leonardo.vasconcelos@email.com",
    "empresaId": 1
  },
  {
    "id": 2,
    "nome": "Pedro Rocha",
    "email": "pedro.rocha@email.com",
    "empresaId": 1
  },
  {
    "id": 3,
    "nome": "Fernando Caetano",
    "email": "fernando.caetano@email.com",
    "empresaId": 2
  }
]
```

Vetor do JavaScript

Lembre-se, jovem:



**“Cada linha do SELECT é convertida para JSON
pelo JavaScript e adicionada a um vetor”**

- Dev Yoda (2024)

Observe o que o Dev Yoda quer dizer

```
SELECT id, nome, email, fk_empresa AS idEmpresa FROM usuario;
```

id	nome	email	idEmpresa
1	Leonardo Vasconcelos	leonardo.vasconcelos@email.com	1
2	Pedro Rocha	pedro.rocha@email.com	1
3	Fernando Caetano	fernando.caetano@email.com	2

SELECT realizado dentro do Workbench

```
[
  {
    "id": 1,
    "nome": "Leonardo Vasconcelos",
    "email": "leonardo.vasconcelos@email.com",
    "empresaId": 1
  },
  {
    "id": 2,
    "nome": "Pedro Rocha",
    "email": "pedro.rocha@email.com",
    "empresaId": 1
  },
  {
    "id": 3,
    "nome": "Fernando Caetano",
    "email": "fernando.caetano@email.com",
    "empresaId": 2
  }
]
```

Vetor com os resultados do SELECT vindo da Model

Agora que temos os dados do usuário e dos aquários de sua empresa em memória, podemos usar esses resultados para preencher a tela inicial do sistema através do objeto “res”.

Criamos o objeto de retorno dentro de “.json()” e colocamos os aquários pertencentes à empresa na chave “aquarios”.

Automaticamente isso retorna um **status 200 (OK)**.

Se não houver aquários encontrados, retornamos o **status 204 (NO CONTENT)**.

```
aquarioModel.buscarAquariosPorEmpresa(resultadoAutenticar[0].empresaId)
  .then((resultadoAquarios) => {
    if (resultadoAquarios.length > 0) {
      res.json({
        id: resultadoAutenticar[0].id,
        email: resultadoAutenticar[0].email,
        nome: resultadoAutenticar[0].nome,
        senha: resultadoAutenticar[0].senha,
        aquarios: resultadoAquarios
      });
    } else {
      res.status(204).json({ aquarios: [] });
    }
  })
```

usuarioController.js

FLUXO

Model

Recebe os dados,
processa no banco e
envia o resultado

Controller

Recebe o resultado da
model e retorna os
dados com o status
HTTP correto

Página

Recebe os dados e
atualiza a tela do
cliente por meio do
JavaScript

FLUXO

Model

Recebe os dados, processa no banco e envia o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

O restante do código lida com diferentes tipos de erros que podem ocorrer durante o processo de login.

Se nenhum usuário for retornado pela consulta, o **status** será **403 (FORBIDDEN)**.

Se houver mais de um usuário com o mesmo login e senha, o código também retorna o mesmo status.

```
} else if (resultadoAutenticar.length == 0) {  
    res.status(403).send("Email e/ou senha inválido(s)");  
} else {  
    res.status(403).send("Mais de um usuário com o mesmo login e senha!");  
}
```

usuarioController.js

FLUXO

Model

Recebe os dados, processa no banco e envia o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

Caso a aplicação tenha seu fluxo quebrado por qualquer outro motivo (falha de credencial no banco, usuário não permitido), é retornado o **status 500 (INTERNAL SERVER ERROR)**.

```
.catch(  
  function (erro) {  
    console.log(erro);  
    console.log("\nHouve um erro ao realizar o login! Erro: ", erro.sqlMessage);  
    res.status(500).json(erro.sqlMessage);  
  }  
);
```

usuarioController.js

Agora voltando para a página de login, onde nasceu a requisição.

Se a resposta da requisição for do tipo **status 200 (OK)**, então alocaremos dentro da **sessionStorage** os dados do nosso usuário e o levamos para a próxima página (pode ser a sua dashboard).

Já já veremos mais sobre sessionStorage...

```
then(function (resposta) {  
  if (resposta.ok) {  
    resposta.json().then(json => {  
      sessionStorage.EMAIL_USUARIO = json.email;  
      sessionStorage.NOME_USUARIO = json.nome;  
      sessionStorage.ID_USUARIO = json.id;  
      sessionStorage.AQUARIOS = JSON.stringify(json.aquarios)  
  
      setTimeout(function () {  
        window.location = "./dashboard/cards.html";  
      }, 1000);  
    });  
  }  
})
```

login.html

FLUXO

Model

Recebe os dados, processa no banco e envia o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

Fim do Processo

Se a resposta não teve o **status 200 (OK)**, então o código irá cair no else e nele trataremos o erro.

Você pode colocar um alerta falando sobre o erro, ou colocar uma div com algum recado explicando o que deu errado.

```
} else {  
  resposta.text().then(texto => {  
    console.error(texto);  
    finalizarAguardar(texto);  
  });  
}
```

login.html

FLUXO

Model

Recebe os dados, processa no banco e envia o resultado

Controller

Recebe o resultado da model e retorna os dados com o status HTTP correto

Página

Recebe os dados e atualiza a tela do cliente por meio do JavaScript

Fim do Processo



FIM DO FLUXO

Mas ficou uma pendência...

O que é a sessionStorage?

A **sessionStorage** permite que o site **armazene dados dentro do navegador** do usuário e possa reutilizar em outras áreas do site **até que ele o feche**. É chamado de memória de sessão.

Segue alguns comandos interessantes sobre essa funcionalidade.

Armazena um novo dado

```
sessionStorage.setItem("chave", "valor");
```

Atribui o dado da chave em uma variável

```
var chave = sessionStorage.getItem("chave"); // retornará "valor"
```

Remove a chave

```
sessionStorage.removeItem("chave");
```

Remove tudo

```
sessionStorage.clear();
```

Referência

[sessionStorage - MDN \(mozilla.org\)](https://developer.mozilla.org/pt-BR/docs/Web/API/SessionStorage)

DICA EXTRA

Você percebeu que as senhas ficam visíveis no banco de dados?

id	nome	email	senha	idEmpresa
1	Leonardo Vasconcelos	leonardo.vasconcelos@email.com	123456	1
2	Pedro Rocha	pedro.rocha@email.com	pedro123	1
3	Fernando Caetano	fernando.caetano@email.com	minha_senha	2

Implementando criptografia ao banco

O MySQL e o Microsoft SQL Server permite que você implemente funcionalidades de criptografia. Ou seja, você pode fazer com que a senha que o usuário inserir seja criptografada antes de ser armazenada no banco.

As criptografias mais comuns são:

Algoritmo	Descrição
MD5	Algoritmo de hash que produz uma saída de 160 bits (20 bytes) geralmente representada por uma string de 40 caracteres hexadecimais.
SHA-1	Inclui variantes com tamanhos de saída de 224, 256, 384, 512 bits. Mais seguro que o SHA-1
SHA-2	É um algoritmo de hash que produz uma saída de 128 bits (16 bytes) geralmente representada por uma string de 32 caracteres hexadecimais.

Se quisermos criptografar com MD5, Basta colocar o valor da senha vinda pelo usuário dentro da função MD5 do MySQL

```
INSERT INTO usuario VALUES  
(NULL, 'Leonardo Vasconcelos', 'leonardo.vasconcelos@email.com', MD5('123456'), 1);
```

id	nome	email	senha	idEmpresa
1	Leonardo Vasconcelos	leonardo.vasconcelos@email.com	e10adc3949ba59abbe56e057f20f883e	1

INSERT realizado no Workbench

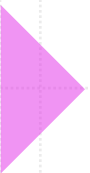
A instrução na camada model deverá ficar da seguinte maneira:

```
var instrucao = `  
  INSERT INTO usuario (nome, email, senha, fk_empresa)  
  VALUES ('${nome}', '${email}', MD5('${senha}'), '${empresaId}');  
`;
```

usuarioModel.js

Podemos comparar se a senha está condizente com a criada pelo usuário da seguinte maneira:

```
SELECT id, nome, email, fk_empresa AS idEmpresa
FROM usuario
WHERE email = 'leonardo.vasconcelos@email.com' AND senha = MD5('123456');
```



id	nome	email	idEmpresa
1	Leonardo Vasconcelos	leonardo.vasconcelos@email.com	1

SELECT realizado no Workbench

A instrução na camada model deverá ficar da seguinte maneira:

```
var instrucao = `
  SELECT id, nome, email, fk_empresa as empresaId
  FROM usuario
  WHERE email = '${email}' AND senha = MD5('${senha}');`;

return database.executar(instrucao);
```

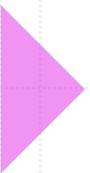
usuarioModel.js



DESAFIOS

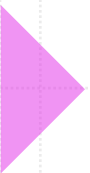
Chegou a hora de adaptar o Web
Data Viz para o seu projeto

Adapte o fluxo de cadastro



O fluxo de cadastro já está pronto no Web Data Viz,
mas você deve alterá-lo para que ele se adapte a
sua regra de negócio

Crie um endpoint para o seu projeto



O usuário quer usar algo interativo e que possa gerar métricas, desenvolva algo que possa ser armazenado no banco de dados

Não esqueça das dicas do Dumbledore:

“Desenvolver uma interface intuitiva será de extrema importância para instigar o interesse do usuário em efetuar seu cadastro”

“Conforme o modelo e suas concepções, forje o script contendo as tabelas que formarão a estrutura de seu banco de dados.”

“É vital que você trace a jornada completa do desenvolvimento, desde a página, passando pela rota, controller e model, para alcançar seus objetivos.”

