

# TRYBE

## Modulo III – Back-end

### Bloco 32: Sockets

#### 1) Arquitetura de Software - Camada de View

MVC – **Model-View-Controller**, com view e sem serviço.  
*Usado por Ruby On Rails e Cake PHP.*

#### O que é MVC?

Essa divisão permite **separar as regras de negócio da interface do usuário.**

Vantagens: Reutilização do código, facilidade de entendimento, facilidade em criar multi interfaces do usuário sem mudar regras do negócio.

Desvantagens: apenas o tempo de planejamento dessas camadas.

MVC é um padrão e não um design pattern, ou seja pode disrespeitar que continua funcionando.

Papel de cada camada:

#### **Model**

Onde acessar, manipular e definir a estrutura dos *dados*. Ex: consulta bd, acessar api.  
As regras de negócio (validações, tratamentos de dados) ficam no model, na ausência de service.  
Deve se manter desacoplada das demais camadas.

#### **View interface**

A camada de apresentação é o que interage com o usuário.

#### **Controller**

Onde receber req e mandar res. Meio de campo entre Model e View (recebe ações da view e decide o que mostrar de volta, após consultar o modelo se necessário).

#### **Router**

Criar rotas (endereço) e comunicar com o Controller.

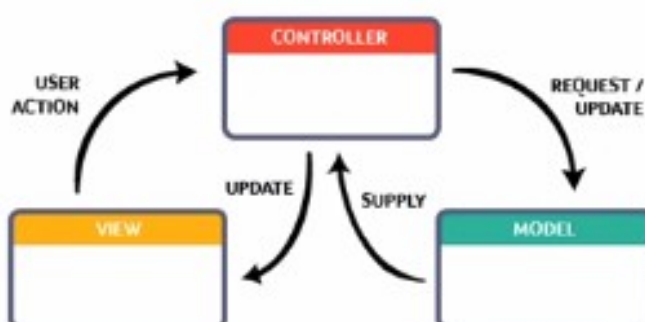
#### View

**Input** de dados: Fornecer meios para que a pessoa possa interagir com o sistema.

**Output** de dados: Cria a visualização dos dados vindos do model.

Formato: em apps web, geralmente é **HTML**, pode também ser JSON e XML.

#### Comunicação entre camadas



## MVC com Express

Criação na mão de uma aplicação em Node e Express usando padrão MVC, segunda as etapas:

- Criar e popular BD (no caso mysql);
- Criar pasta do app, inicializar node, instalar mysql2 e express;
- Criar model/connection.js
- Criar bases de model e controller do mesmo tema
- Uso para a view de [template engine](#) (criar documentos, inclusive HTML, dinamicamente, integrando code em arquivos – no caso [EJS](#)).

```
<!-- <!doctype html>
<html>
  <head>
    <title>MVC - Exemplo</title>
  </head>
  <body>
    <ul>
      <% authors.forEach((author) => { %>
        <li><%= author.name %></li> -->
        <a href=<%= ' /authors/${author.id}' %>>Ver detalhes</a>
      <!-- <% } %>
    </ul>
  </body>
</html> -->
```

Escritura <%= %>

```
<ul>
  <% authors.forEach((author) => { %>
    <li><%= author.name %></li>
  <% } %>
</ul>
```

- No index.js, todo o normal além de setar EJS

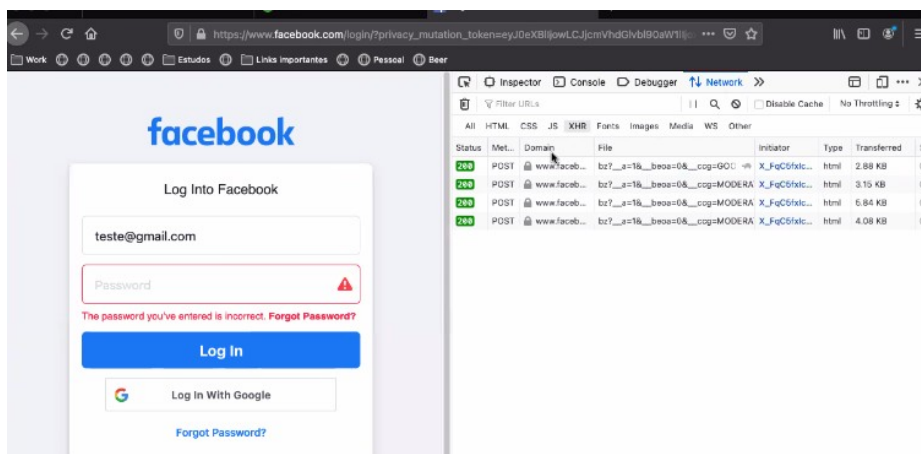
```
app.set('view engine', 'ejs');
app.set('views', './views');
```

- Escritura da response no controller do EJS

```
res.render('authors/index', { authors });
```

- Continuar para cada feature o fluxo View (o que mostra) Controller (functions para req e res), Model (interagir com bd) e endpoints no index.js .

## Dicas diversas:



Console – Network no navegador para visualizar a requisição feita a partir de nossa ação de usuário.

## 2) Sockets - TCP/UDP & NET

### Def

**Mecanismo de comunicação entre máquinas através da rede**, usando o protocolo **TCP/IP**.

Uso do pacote **NET** do Node.js para criar aplicações que trafeguem mensagens através de sockets.

### Como sockets funcionam?

Sempre tem lado server e lado cliente.

Socket usados quando algo tem que se manter: exemplos de chat & alerta.

Dois tipos de sockets:

- socket **stream**, tipicamente implementados via TCP;
- socket **dgram**, tipicamente implementados via UDP.

### Modelo OSI vs TCP

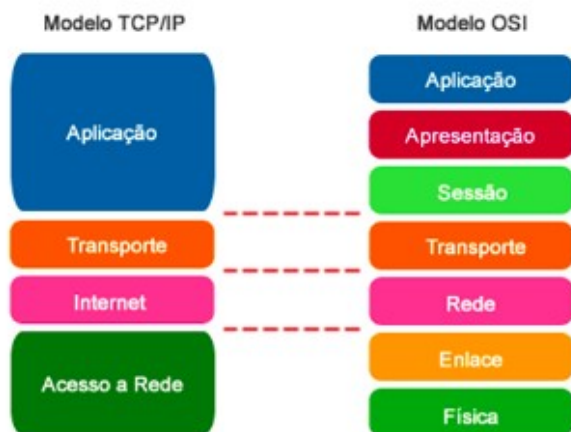
#### Modelo OSI (Open System Interconnection)

Modelo de rede de computador, referência da ISO (*Internacional Organization for Standardization*), dividido em **7 camadas** de papéis que são ações necessárias para que ocorra a **interconectividade dos dispositivos**:



**Encapsulamento de 7 para 1, desencapsulamento de 1 para 7** - bit voltam a se tornar dados.

## Modelo TCP/IP



// Agrupamento de 1,2 e 5,6,7 para somar 4 camadas.

## TCP e UDP

Portas TCP e UDP. Exemplo: localhost:300 é protocolo HTTP + o endereço da nossa máquina, o localhost ou 127.0.0.1 + nossa porta 3000 .

### TCP

“Pré-acordo” entre cliente e servidor chamado **Three Way Handshake** ( SYN , SYN-ACK , ACK ). →

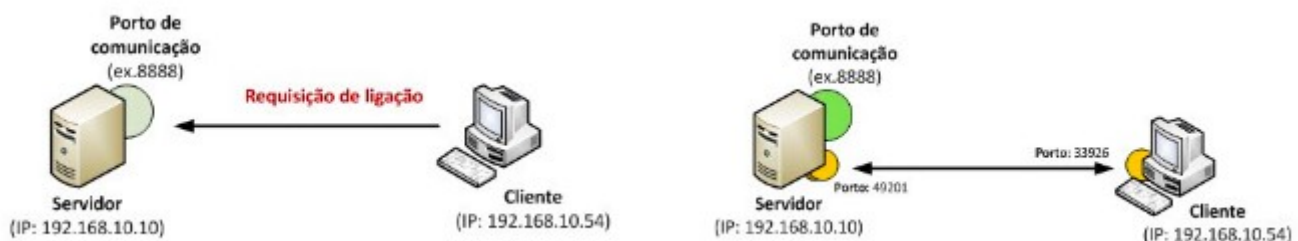
- Cliente: **SYN** Quero me conectar!
- Servidor: **ACK** Mensagem recebida!
- Servidor: **SYN** Vamos nos conectar!
- Cliente: **ACK** Mensagem recebida!

### UDP

Mais simples, **sem garantia** na entrega dos pacotes.  
Máquina emissora envia uma determinada informação para máquina receptora, sem confirmação em troca.

## Sockets TCP

Def: *abstração para endereços* de comunicação através dos quais as máquinas se comunicam.  
Usando **endereço único (IP)** para identificar máquinas + **porta** para identificar aplicações.



1. **Cliente solicita conexão** ao servidor (que está em loop esperando ligações)
2. **Servidor aceita e gera socket**, liberando porta inicial

## No code: como implementar e transferir dados via TCP

→ Escrever ambos lados server e client

projeto/server.js

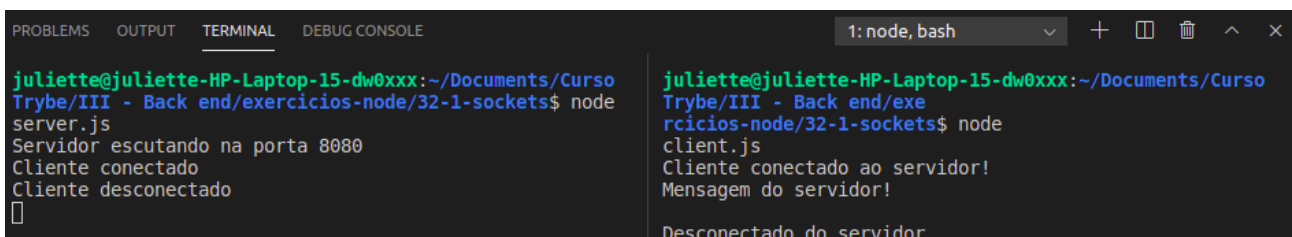
```
/* Importando o pacote NET, responsável pela implementação dos sockets no Node. */
const net = require('net');
/* Criando o servidor com o método 'createServer', onde recebe uma conexão na qual são expostos
os eventos que podemos manipular no nosso servidor. */
const server = net.createServer((connection) => {
  console.log('Cliente conectado');
  /* Assim como um evento normal do Node.js, o método ".on()" escuta um evento em específico e,
quando ele é ativado, nossa função de callback é chamada. */
  connection.on('end', () => {
    console.log('Cliente desconectado');
  });
  /* Nessa conexão que foi aberta, podemos fazer várias coisas. Uma delas é escrever/devolver uma
mensagem para o cliente. */
  connection.write('Mensagem do servidor!\r\n');
  connection.pipe(connection);
});
/* Após termos programado o servidor, é só colocá-lo de pé */
server.listen(8080, () => {
  console.log('Servidor escutando na porta 8080');
});
```

projeto/client.js

```
const net = require('net');
/* Através do pacote NET, nós podemos não só criar servidores como podemos conectar nossos
clientes aos servidores */
const client = net.connect({ port: 8080 }, () => {
  console.log('Cliente conectado ao servidor!');
});
/* Assim como no servidor, também temos eventos do lado do cliente, onde o evento 'data' é ativado
quando o servidor envia uma mensagem para o cliente. */
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
/* Quando a conexão é interrompida/terminada, é ativado o evento 'end', onde podemos limpar
alguns caches, dar uma mensagem para usuário, atualizar algum dado no banco de dados etc. */
client.on('end', () => {
  console.log('Desconectado do servidor');
});
```

→ Usar **eventos do pacote net** : close, connect, data, drain, end, error, lookup, ready, timeout.

→ Executar de ambos lados via `node server.js` e `node client.js` .



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
1: node, bash
juliette@juliette-HP-Laptop-15-dw0xxx:~/Documents/Curso
Trybe/III - Back end/exercicios-node/32-1-sockets$ node
server.js
Servidor escutando na porta 8080
Cliente conectado
Cliente desconectado
[]

juliette@juliette-HP-Laptop-15-dw0xxx:~/Documents/Curso
Trybe/III - Back end/exe
rcios-node/32-1-sockets$ node
client.js
Cliente conectado ao servidor!
Mensagem do servidor!
Desconectado do servidor
```

## Dicas diversas

[Socket de chat](#) na aula ao vivo.

---

### 3) Sockets – Socket.io

NET pode ser custoso para app *real time* de grande porte. Socket.io é mais simples e scalable.

#### O que é socket.io

Antigamente real time era feito via **pooling** ou seja loop infinito ficando para verificar algo.

Vantagens agora com socket.io:

- comunica via sockets;
- tem **fallBack**, feature de contingência para quando client/server não estiver disponível;
- funciona via **eventos Node**, podendo ouvir evento de conexão e dispara function a partir disso.

#### Processo para implementar no code

- \* npm init -y
- \* npm install express
- \* Escrever endpoint de GET na porta 3000 no index.js
- \* Escrever content (aparência de chatbot) no index.html
- \* npm install socket.io cors
- \* integrar ele no index.js

```
const app = require('express')();
const http = require('http').createServer(app);
const cors = require('cors');
const io = require('socket.io')(http, {
  cors: {
    origin: 'http://localhost:3000', // url aceita pelo cors
    methods: ['GET', 'POST'], // Métodos aceitos pela url
  }
});
app.use(cors()) // Permite recursos restritos na página web serem pedidos a domínio externo
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});
io.on('connection', (socket) => {
  console.log(
    'Usuário conectado, igual ao que fizemos na aula anterior, porém dessa vez em um servidor escalável'
  );
});
http.listen(3000, () => {
  console.log('Servidor ouvindo na porta 3000');
});
```

\*integrar ele no index.html com o link CDNJS (CDN seria "/socket.io/socket.io.js")

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/3.0.4/socket.io.js"></script>
```

```
<script>
```

```
  const socket = io();
```

//objeto io é global e, assim que chamado, executa uma conexão socket com alguém.

Tem endpoint por onde é acessado como param, ex: const socket = io('http://localhost:5000');

```
</script>
```

\*Gerir connection e disconnection no index.js

```
io.on('connection', (socket) => {  
  console.log('Conectado');  
  socket.on('disconnect', () => {  
    console.log('Desconectado');  
  });  
});
```

\*rodar com node index.js no terminal e interagir no browser via localhost:3000

\*criar no html eventlistener e usar **"escuta de evento"** `.on()` para sair do terminal puro

\*manipular .js com propriedades como:

**emit** (para mandar para o user)

```
io.emit('Nome do seu evento', {  
  propriedade: 'Do seu objeto',  
  enviado: 'Para o cliente da conexão atual'});
```

**broadcast** (para mandar para todos outros users)

```
socket.broadcast.emit('mensagemServer', { mensagem: ' Iiiiiirraaaa! Fulano acabou de se conectar :D'});
```

Code final:

index.html

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/3.0.4/socket.io.js"></script>  
<script>  
  const socket = io();  
  const form = document.querySelector('form')  
  const inputMessage = document.querySelector('#mensagemInput')  
  form.addEventListener('submit', (e) => {  
    e.preventDefault();  
    socket.emit('mensagem', inputMessage.value);  
    inputMessage.value = ''  
    return false;  
  });  
  // cria uma `li` e coloca dentro da `ul` com `id` mensagens  
  const createMessage = (message) => {  
    const messagesUl = document.querySelector('#mensagens');  
    const li = document.createElement('li');  
    li.innerText = message;  
    messagesUl.appendChild(li);  
  }  
  // Quando nosso evento `ola` for emitido, vamos pegar a string mensagem enviada pelo nosso evento e passar para a função `createMessage`  
  socket.on('ola', (mensagem) => createMessage(mensagem));  
  // Aqui o evento é recebido da mesma maneira do último, mas este não é recebido pelo cliente que enviou o mesmo  
  socket.on('mensagemServer', (objeto) => createMessage(objeto.mensagem));  
</script>
```

index.js

```
io.on("connection", (socket) => {  
  console.log('Conectado');  
  
  socket.emit('ola', 'Bem vindo fulano, fica mais um cadin, vai ter bolo :)');  
  
  socket.broadcast.emit('mensagemServer', {  
    mensagem: ' Iiiiiirraaaa! Fulano acabou de se conectar :D'});  
  
  socket.on('disconnect', () => {  
    console.log('Desconectado');  
  });  
  
  socket.on('mensagem', (msg) => {  
    io.emit('mensagemServer', { mensagem: msg });  
  });  
});
```

// consegue fazer várias salas

Dicas diversas da aula ao vivo

Recado: socket.io não é implementação do websocket



```
const socketIoServer = require('http').createServer()
const io = require('socket.io')(socketIoServer, {
  cors: {
    origin: 'http://localhost:3000',
    methods: ['GET', 'POST']
  }
})
```

server.js

servidor que nomeamos socketIoServer, ele que ouve porta no final.

Diz que apenas pode receber req dessa origem com esses methods.

[Link do code](#) da aula ao vivo.

## 4) Projeto – Webchat

### Dicas diversas do projeto

#### **Recursos**

Doc oficial socket.io:

<https://socket.io/docs/v3/index.html> & <https://github.com/socketio/socket.io> & tutorial sobre chat <https://socket.io/get-started/chat>

Aula de revisão da T4, arq. similar: <https://github.com/tryber/sd-04-live-lectures/pull/67/files>

—

#### **http**

express tem um http dentro - socket não, por isso precisa do recurso de http e createServer

—

#### **Comunicar entre Server & Client**

“sending an event is done with: *socket.emit()*

receiving an event is done by registering a listener: *socket.on(<event name>, <listener>)*”

Ou seja

.emit é emitir mensagem

.on é reagir a receber isso

—

#### **Como testar o lado server?**

- com testes do projeto

- com client (por isso vontade de desenvolver req1&2 junto)

—

#### **Dar nome aleatório**

- **faker** npm package <https://www.npmjs.com/package/faker> to get random names about anything

- formula tipo `User${Math.round(Math.random() * 1000)}`

—

#### **Lidar com testes**

- testar req 4 a realizar sem browser aberto para evitar adicionar users no teste assim quebrar ele

- erro de Timeout: verificar conexão com banco, data-testids, funções assíncronas, ordem funções

—

#### **Adicionar elemento no topo**

- no Js, push no começo de array com **unshift** [https://www.w3schools.com/jsref/jsref\\_unshift.asp](https://www.w3schools.com/jsref/jsref_unshift.asp)

- no dom, **prepend** <https://developer.mozilla.org/fr/docs/Web/API/ParentNode/prepend>



## Bloco 33 – Projeto Trybeer II

### Dicas diversas do projeto

#### **Socket**

- No index.js, criar server para que ambos possam rodar na mesma porta.

```
const server = require('http').createServer(app);
```

- O lado do client pode colocar como component React que renderiza o que precisa ser visto dos dois lados.