

INTENSIVÃO DE JAVASCRIPT

Apostila Completa Aula 4

Guia passo a passo: Chat em Tempo Real



Parte 1

Instalando as ferramentas





Instalando as ferramentas – VS Code X Atom

Durante o intensivão, o professor Daniel vai usar duas ferramentas para organizar nosso trabalho: o VS Code para o Back-end e o Atom para o Front-end.

O VS Code e o Atom são ambos editores de código que ajudam os desenvolvedores a escrever e gerenciar seus projetos. Eles possuem recursos semelhantes, como realce de sintaxe, autocomplete e integração com controle de versão.

No entanto, muitos desenvolvedores preferem o VS Code por algumas razões:

Integração e Suporte da Comunidade: O VS Code é desenvolvido pela Microsoft e possui um grande suporte da comunidade. Isso significa que há muitas extensões disponíveis, suporte aprimorado para diferentes linguagens de programação e uma ampla documentação online.

Desempenho e Velocidade: O VS Code é conhecido por ser rápido e eficiente, mesmo em projetos grandes. Ele oferece uma experiência de edição de código suave e responsiva.

Integração com Ferramentas: O VS Code possui integração nativa com ferramentas populares de desenvolvimento, como Git, GitHub, depuradores e terminais integrados. Isso facilita o fluxo de trabalho do desenvolvedor, tornando-o mais produtivo.

Portanto, embora o Atom seja uma opção viável, o professor Daniel sugere o uso do VS Code devido às suas vantagens em termos de desempenho, suporte da comunidade e integração com outras ferramentas de desenvolvimento.

No próximo slide, você encontrará informações sobre o Atom e como instalá-lo, mas você não precisa usá-lo para criar seu projeto de Chat em Tempo Real.

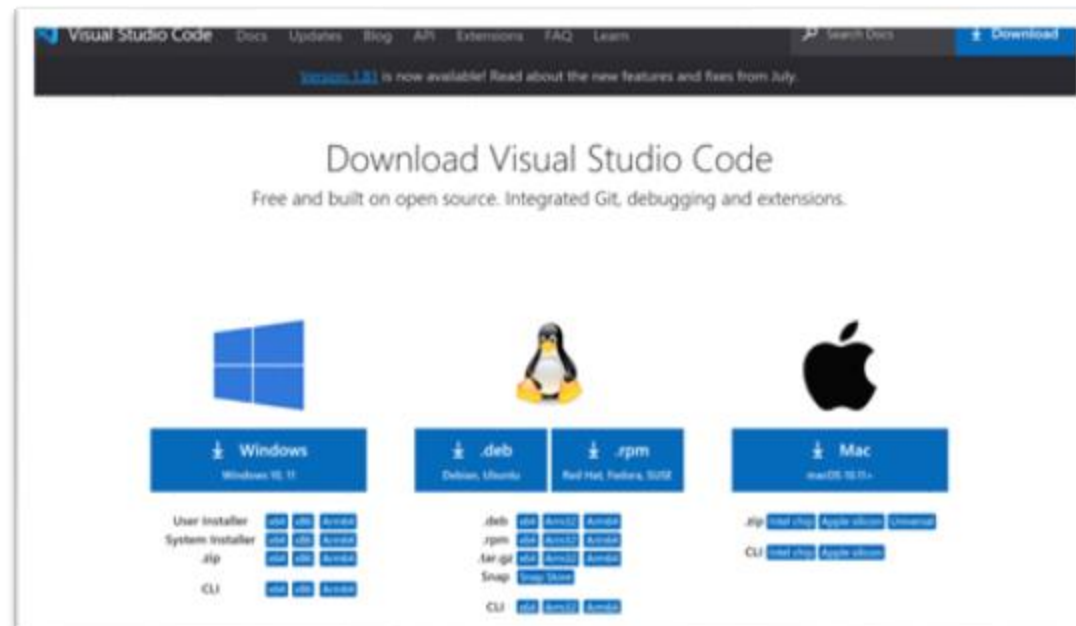


Instalando as ferramentas – VS CODE

Caso você ainda não tenha o Visual Studio Code instalado, basta seguir os procedimentos abaixo. A instalação do VS Code é totalmente gratuita, e você pode usá-lo em seu computador sem precisar pagar nada. O link para fazer o download do programa é mostrado abaixo:

<https://code.visualstudio.com/>

- 1 – Baixe o arquivo de instalação correspondente ao seu sistema operacional (Windows, MacOS ou Linux).
- 2 – Execute o arquivo de instalação e siga as instruções na tela. Em geral, é só continuar clicando em "Próximo".
- 3 – No Windows, durante a instalação, marque a opção "Add to path" para adicionar o VS Code às suas variáveis de ambiente.





Instalando as ferramentas – Atom (Opcional)

O Atom é um editor de texto de código aberto desenvolvido pela GitHub. Ele é projetado para ser altamente personalizável e adaptável às necessidades dos desenvolvedores. Com recursos como realce de sintaxe, autocomplete e integração com controle de versão, o Atom é popular entre programadores para escrever código em diversas linguagens de programação.



Passo a passo para baixar o Atom:

1- Acesse o Site Oficial: Vá para o site oficial do Atom em atom.io.

2- Escolha o Sistema Operacional: Na página inicial, você verá opções para baixar o Atom para diferentes sistemas operacionais, como Windows, macOS e Linux. Clique na opção que corresponde ao seu sistema.

2-Download do Instalador: Após selecionar o sistema operacional, o site irá redirecioná-lo para a página de download específica para o seu sistema. Clique no botão de download para iniciar o processo de obtenção do instalador do Atom.

3-Instalação: Depois de baixar o instalador, localize o arquivo em seu computador e execute-o. Siga as instruções na tela para completar o processo de instalação.

4-Executar o Atom: Depois de instalado, você pode executar o Atom a partir do menu de aplicativos do seu sistema operacional.

Uma vez que o Atom esteja instalado, você pode começar a utilizá-lo para escrever e editar código em uma variedade de linguagens de programação. Ele oferece uma série de recursos e extensões que podem ser personalizados para atender às suas necessidades de desenvolvimento.

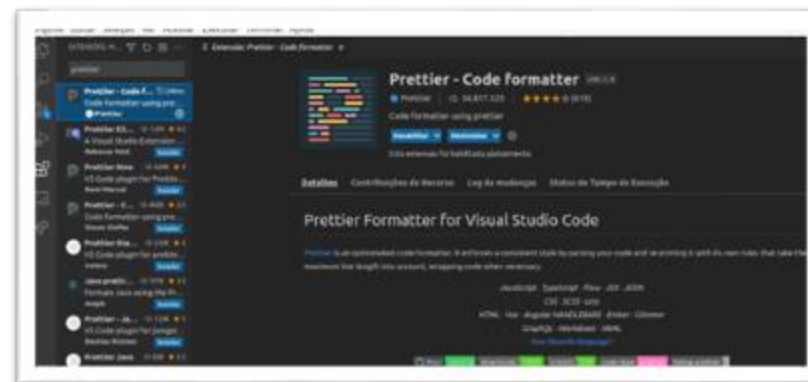


Instalando as ferramentas – VS CODE - EXTENSÕES

As extensões no Visual Studio Code são pequenos programas que adicionam recursos extras ao editor de código. Elas são como "plugins" que podem ser instalados para personalizar e estender as funcionalidades do VS Code.

A extensão Prettier adiciona a capacidade de formatar automaticamente o código, tornando-o mais legível e organizado.

Para instalar a extensão Prettier no Visual Studio Code:



- Clique no ícone de extensões no menu lateral esquerdo (ou use o atalho "Ctrl+Shift+X").
- Na barra de pesquisa, digite "Prettier".
- A extensão "Prettier - Code Formatter" deve aparecer nos resultados da pesquisa. Clique no botão "Instalar" ao lado dela.
- Após a instalação, você pode configurar o Prettier como o formatador padrão para o seu projeto. Para fazer isso, abra as configurações do Visual Studio Code (menu "File" > "Preferences" > "Settings" ou use o atalho "Ctrl+,").
- Na barra de pesquisa das configurações, digite "Default Formatter" e selecione a opção "Editor: Default Formatter".
- No campo de valor, digite "esbenp.prettier-vscode" e pressione Enter para salvar as alterações.
- Agora, o Prettier está instalado e configurado como o formatador padrão no Visual Studio Code. Você pode usar o comando "Format Document" (menu "Edit" > "Format Document" ou atalho "Shift+Alt+F") para formatar o código.



Instalando as ferramentas – Node.js

O Node.js é um ambiente de execução de código JavaScript do lado do servidor. Ele permite que você execute código JavaScript fora do navegador, o que significa que você pode criar aplicativos de servidor, scripts de linha de comando e muito mais usando JavaScript. Para instalar o Node.js, você pode seguir os seguintes passos:



- Acesse o site oficial do Node.js em <https://nodejs.org>.
- Na página inicial, você verá duas versões para download: LTS (Long Term Support) e Current. A versão LTS é recomendada para a maioria dos usuários, pois é mais estável e possui suporte a longo prazo. Selecione a versão LTS ou a versão mais recente, se preferir.
- Após selecionar a versão desejada, você será redirecionado para a página de download. Escolha o instalador adequado para o seu sistema operacional (Windows, macOS ou Linux) e clique no link para iniciar o download.
- Após o download ser concluído, execute o instalador e siga as instruções na tela para concluir a instalação.
- Após a instalação ser concluída, você pode verificar se o Node.js foi instalado corretamente abrindo o terminal ou prompt de comando e digitando o comando `node -v`. Se a versão do Node.js for exibida, significa que a instalação foi bem-sucedida.

Parte 2

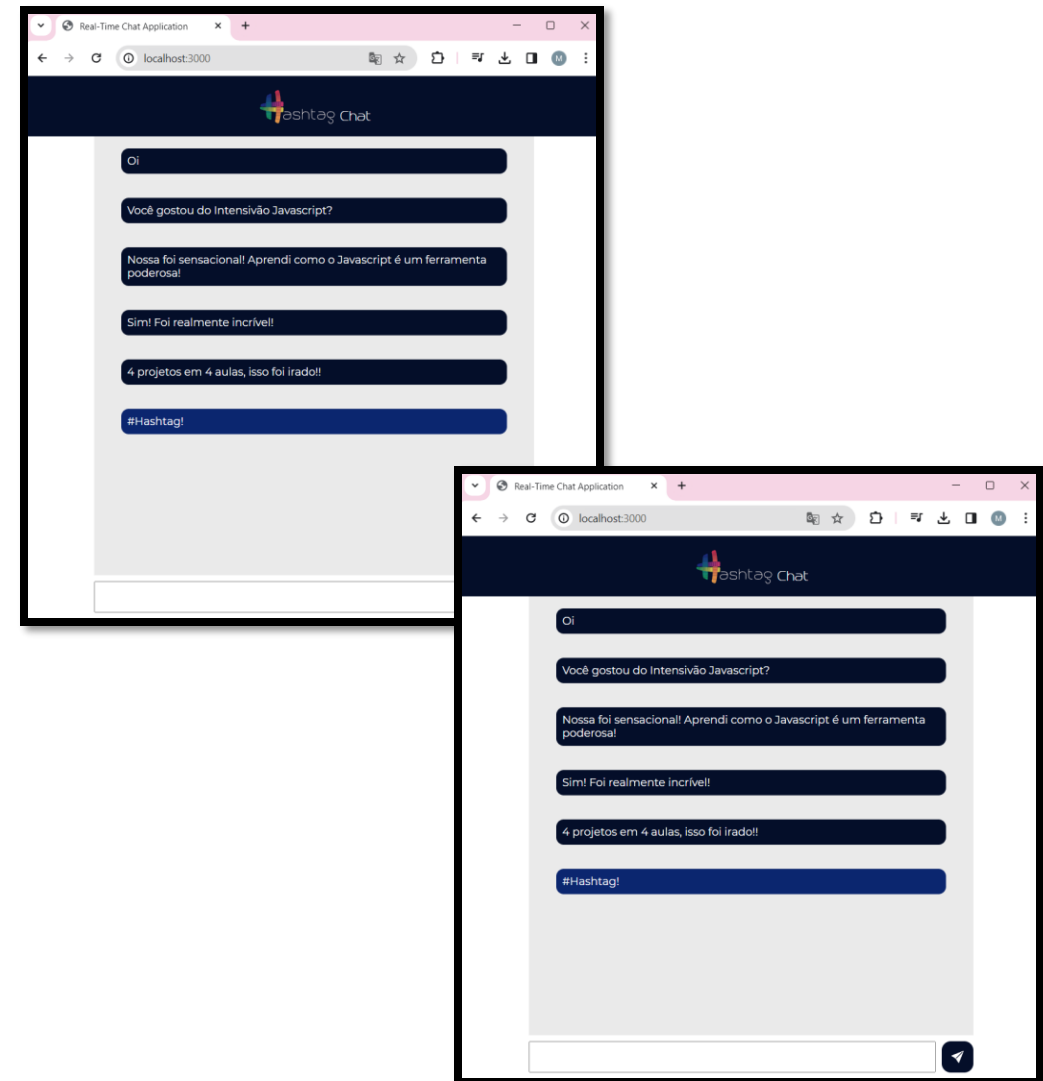
Apresentação do Projeto Chat em Tempo Real!

Apresentação do Projeto Chat em Tempo Real!

Bem-vindos ao **Chat em Tempo Real**, onde exploraremos a implementação da lógica utilizando JavaScript tanto no frontend quanto no backend. Nosso objetivo é oferecer uma plataforma interativa para comunicação instantânea entre usuários, tornando as conversas simples e acessíveis.

Dentro deste guia, você encontrará um passo a passo detalhado para implementar a lógica do nosso Chat em Tempo Real, desde a configuração do frontend até a criação do servidor backend utilizando Node.js e Socket.io para comunicação em tempo real entre os usuários.

Estamos empolgados para criar uma plataforma de comunicação instantânea envolvente e convidativa. Prepare-se para uma experiência de conversação deliciosa e inovadora!



Parte 3

Front-end X Back-end

Front-end X Back-end

Projetos Frontend e Projetos Backend em JavaScript são duas áreas distintas do desenvolvimento web que trabalham juntas para criar aplicativos web dinâmicos e interativos.

Projetos Frontend:

O Frontend refere-se à parte da aplicação web com a qual os usuários interagem diretamente. Isso inclui o layout, design, e interatividade que o usuário vê e com o qual interage em um navegador da web.

- **Tecnologias Frontend em JavaScript:** No frontend, JavaScript é a linguagem principal para tornar as páginas web interativas. Junto com HTML (HyperText Markup Language) e CSS (Cascading Style Sheets), o JavaScript é fundamental para criar uma experiência de usuário envolvente e responsiva.
- **Frameworks e Bibliotecas:** Existem muitos frameworks e bibliotecas JavaScript populares para o frontend, como React.js, Angular.js e Vue.js. Essas ferramentas ajudam os desenvolvedores a criar interfaces de usuário complexas e dinâmicas de forma mais eficiente.
- **Responsividade e Design:** No frontend, os desenvolvedores se concentram na responsividade do design, garantindo que o aplicativo funcione bem em diferentes dispositivos e tamanhos de tela.

Front-end X Back-end

Projetos Backend:

O Backend é a parte invisível de um aplicativo web que lida com a lógica de negócios, processamento de dados e interações com o servidor. Ele suporta e gerencia o funcionamento do frontend.

- **Tecnologias Backend em JavaScript:** Node.js é a tecnologia principal para desenvolvimento backend em JavaScript. Ele permite que os desenvolvedores usem JavaScript no servidor, o que simplifica a comunicação entre frontend e backend, já que a mesma linguagem pode ser usada em ambos os lados.
- **Frameworks e Bibliotecas:** Existem diversos frameworks e bibliotecas para o desenvolvimento backend em JavaScript, como Express.js, Nest.js, e Koa.js. Essas ferramentas fornecem estruturas e funcionalidades para criar servidores web robustos e escaláveis.
- **Gerenciamento de Dados:** No backend, os desenvolvedores lidam com operações de banco de dados, autenticação de usuários, manipulação de arquivos e outras tarefas relacionadas ao armazenamento e recuperação de dados.

Em resumo, enquanto o Frontend se concentra na experiência do usuário e na interface visual, o Backend lida com a lógica de negócios e a manipulação de dados nos bastidores. Ambos são essenciais para o desenvolvimento de aplicativos web modernos e funcionam em conjunto para fornecer uma experiência completa aos usuários.

Front-end X Back-end

Projetos FullStack:

Um projeto fullstack de JavaScript é aquele que engloba tanto o desenvolvimento do frontend quanto do backend utilizando JavaScript como linguagem principal. Nesse tipo de projeto, JavaScript é utilizado em todas as camadas da aplicação web, desde a interface do usuário até a lógica de servidor e o banco de dados, se aplicável.

Componentes de um Projeto Fullstack em JavaScript:

Frontend:

- Responsável pela interface do usuário (UI) que os usuários interagem diretamente em um navegador web.
- Utiliza tecnologias como HTML, CSS e JavaScript para criar uma experiência interativa e responsiva.
- Frameworks como React.js, Angular.js, Vue.js são comuns para construção de interfaces dinâmicas.

•Backend:

- Lida com a lógica de negócios, processamento de dados e interações com o servidor.
- JavaScript pode ser usado no servidor com o auxílio de plataformas como Node.js, permitindo que os desenvolvedores usem uma linguagem comum em todo o projeto.
- Frameworks como Express.js, Nest.js, e Koa.js são comumente usados para criar APIs RESTful e gerenciar rotas no servidor.

Parte 4

Primeiros passos



Primeiros Passos



Para iniciar o nosso projeto, vamos criar uma nova pasta com o nome "Chat-Treino-Copia". Essa pasta será o local onde iremos armazenar todos os arquivos do nosso Chat em Tempo Real. Dentro dessa pasta, colocaremos os arquivos que serão utilizados para estilização, arquivos adicionais (imagens, cardápio) e configuração do ambiente de desenvolvimento.

Vou te explicar passo a passo como criar uma pasta nova no Windows:

- Primeiro, abra o "Explorador de Arquivos" clicando no ícone da pasta amarela na barra de tarefas ou pressionando a tecla "Windows" + "E" no teclado.
- Navegue até o local onde você deseja criar a nova pasta. Por exemplo, você pode escolher criar a pasta na área de trabalho ou dentro de outra pasta existente.
- Com o local selecionado, clique com o botão direito do mouse em um espaço vazio da janela do Explorador de Arquivos. Um menu de opções será exibido.

Primeiros Passos

- No menu de opções, passe o cursor sobre a opção "Novo" e, em seguida, clique em "Pasta". Uma nova pasta será criada com o nome "Nova pasta" destacado.
- Digite o nome desejado para a nova pasta. No nosso caso, digite "Chat-Treino-Copia" como o nome da pasta.
- Pressione a tecla "Enter" no teclado para confirmar o nome da pasta. A nova pasta será criada no local selecionado.

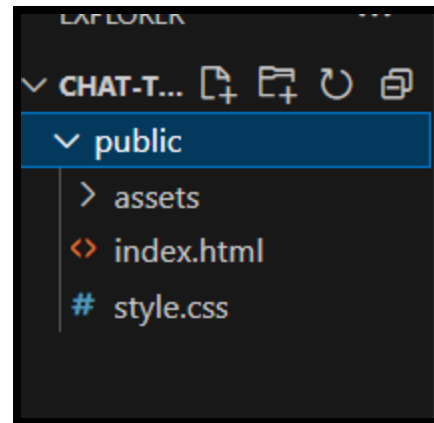
Nome	Status	Data de modificação	Tipo	Tamanho
 Chat-Treino-Copia		31/01/2024 17:54	Pasta de arquivos	

Primeiros Passos

Nesse momento, você fará o download do arquivo disponível:

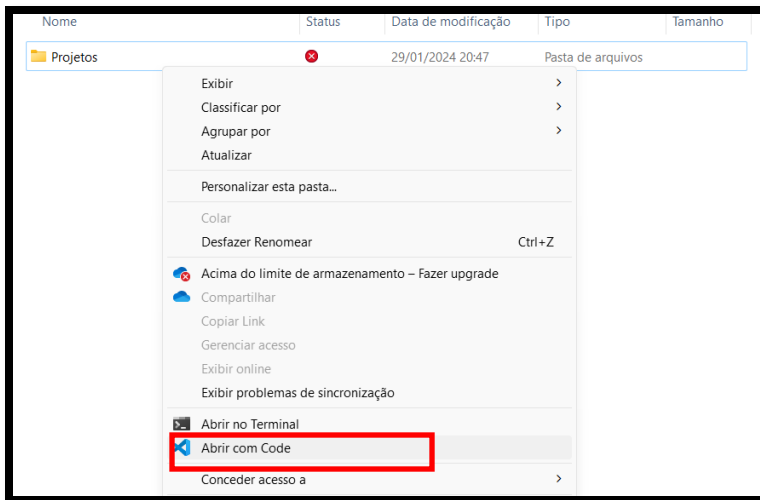


Adicione o arquivo "public" à pasta "Chat-Treino-Copia" do nosso projeto.

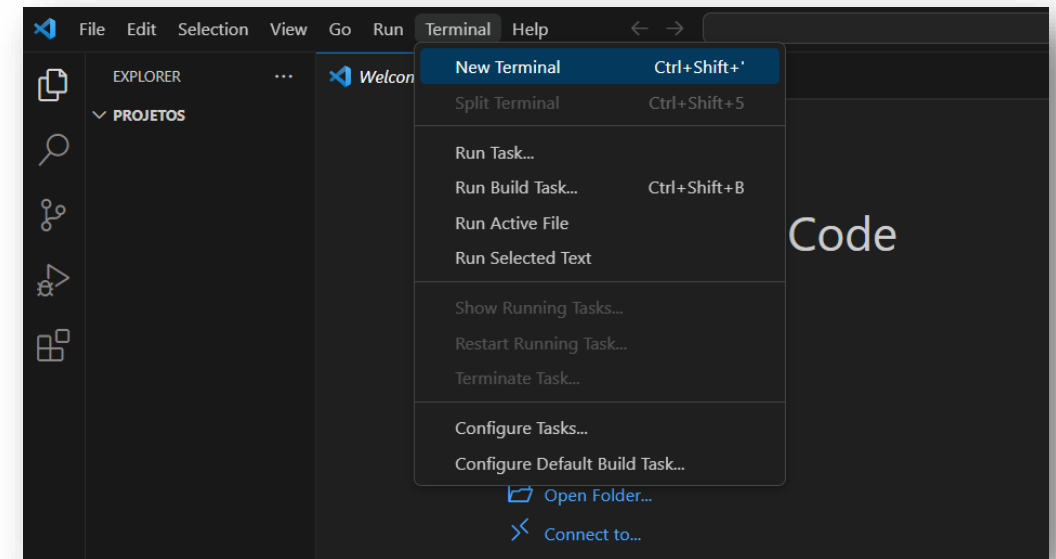


Primeiros Passos

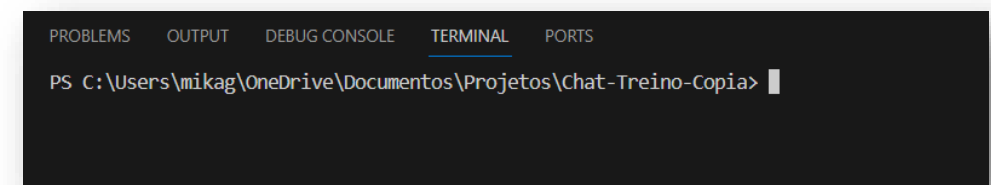
1 - Dentro da pasta recém criada "Chat-Treino-Copia", você irá clicar com o botão direito do mouse e abrir a opção "Abrir com Code".



2 - Abra o terminal do VS Code, clicando na aba Terminal e depois em "New Terminal"(Novo Terminal):



E um terminal irá abrir na parte inferior do VS Code:



Primeiros Passos

Terminal:

Dentro do Visual Studio Code (VS Code), um terminal é uma interface de linha de comando integrada que permite interagir com o sistema operacional diretamente dentro do ambiente de desenvolvimento. Isso significa que você pode executar comandos diretamente no terminal sem precisar sair do VS Code.

Os terminais no VS Code podem ser personalizados para usar diferentes shells, como o PowerShell no Windows, o Terminal no macOS e o Bash ou Zsh no Linux, entre outros. Você pode abrir vários terminais dentro do VS Code, cada um em uma aba separada, o que é útil para executar várias tarefas simultaneamente ou para trabalhar em diferentes partes do seu projeto.

Os comandos que você coloca no terminal são basicamente instruções que você dá ao sistema operacional para executar uma determinada tarefa. Por exemplo, você pode usar comandos para compilar seu código, gerenciar arquivos e pastas, iniciar servidores locais, instalar dependências, entre outras tarefas relacionadas ao desenvolvimento de software.

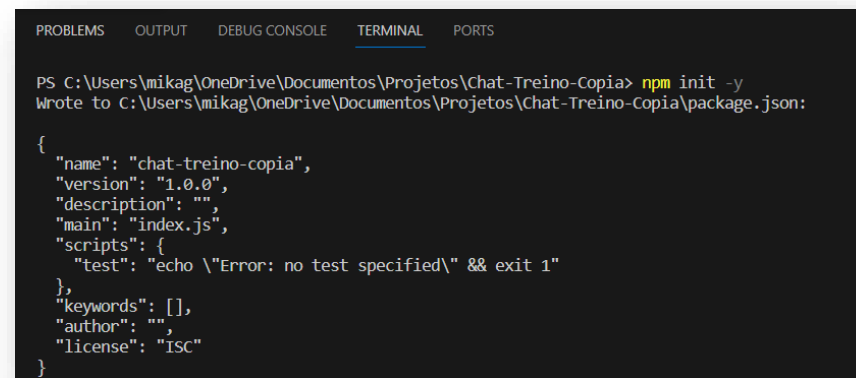
Primeiros Passos

Agora com o terminal aberto, vamos executar o nosso primeiro comando que será "npm init -y". O comando **npm init -y** é utilizado para inicializar um novo projeto Node.js de forma rápida e automática, criando um arquivo **package.json** com configurações padrão.

Aqui está o que cada parte do comando faz:

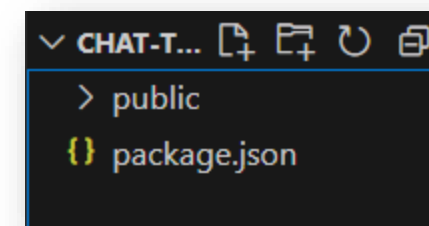
- **npm**: é o Node Package Manager, uma ferramenta que facilita a instalação e gerenciamento de pacotes JavaScript.
- **init**: é um subcomando do **npm** utilizado para iniciar um novo projeto.
- **-y**: é uma opção que responde automaticamente "sim" a todas as perguntas feitas durante a inicialização do projeto. Isso significa que o **npm** criará o arquivo **package.json** com valores padrão para todas as configurações, sem exigir entrada do usuário.

Em resumo, **npm init -y** inicializa um novo projeto Node.js de forma rápida e automatizada, criando um arquivo **package.json** com valores padrão para todas as configurações. Isso é útil quando você quer começar rapidamente um novo projeto e não precisa personalizar detalhes específicos na configuração inicial.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\mikag\OneDrive\Documentos\Projetos\Chat-Treino-Copia> npm init -y
Wrote to C:\Users\mikag\OneDrive\Documentos\Projetos\Chat-Treino-Copia\package.json:

{
  "name": "chat-treino-copia",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```



```
▼ CHAT-T... [Icons]
  > public
  {} package.json
```

Primeiros Passos

Vamos criar um arquivo chamado "servidor.js" para programar a parte de back-end dentro dele. E antes de adicionarmos alguma linha de código dentro dele, precisamos entender um pouco sobre a comunicação da internet e algumas estruturas.

A arquitetura da internet é composta por uma série de elementos interligados que trabalham juntos para fornecer a infraestrutura que possibilita a navegação na web. Em essência, a internet é uma gigantesca rede de computadores que se comunicam entre si. Essa comunicação é possível graças a uma série de protocolos e sistemas, como o Protocolo de Internet (IP), o Sistema de Nomes de Domínio (DNS) e o Protocolo de Transferência de Hipertexto (HTTP).

1 - Endereços IP e DNS: Cada computador conectado à internet tem um endereço IP único, que é uma série de números usada para identificar o computador na rede. No entanto, lembrar esses números pode ser difícil para os humanos, então usamos nomes de domínio (como google.com) para facilitar a identificação dos sites. O DNS é o sistema que traduz esses nomes de domínio em endereços IP.

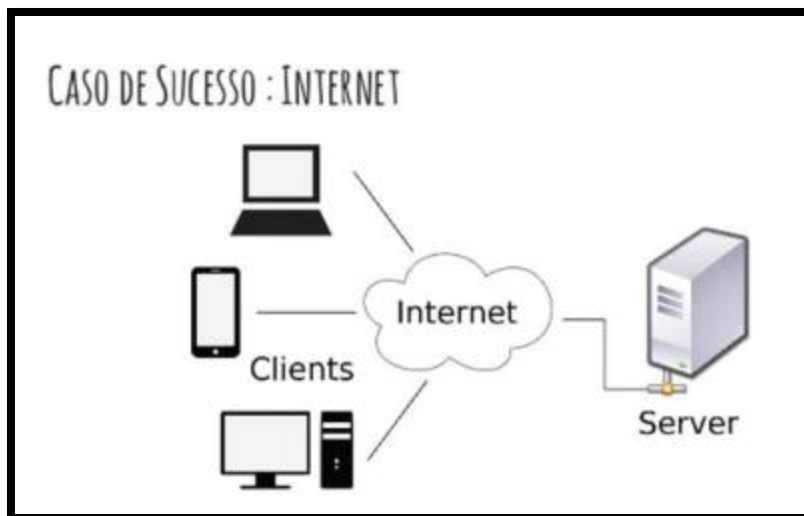
2 - HTTP e servidores web: Quando você digita um endereço de site no seu navegador, o navegador envia uma requisição HTTP para o servidor que hospeda o site. Se o servidor aprovar a requisição, ele envia os arquivos do site para o navegador em pequenos pedaços chamados pacotes de dados. O navegador então monta esses pedaços para mostrar o site completo.

3 - TCP/IP: O HTTP e todas as outras informações enviadas entre o cliente (seu computador) e o servidor são transmitidas através da sua conexão de internet usando o TCP/IP. O TCP/IP é um conjunto de regras que definem como os dados devem ser enviados e recebidos na internet

Primeiros Passos

A arquitetura da web é uma parte importante da arquitetura da internet, pois ela define como as informações são organizadas e apresentadas nos sites. Uma boa arquitetura da web facilita a navegação dos usuários e ajuda a melhorar a experiência do usuário. Essa é uma visão geral simplificada da arquitetura da internet. Na realidade, há muitos outros componentes e protocolos envolvidos, cada um desempenhando um papel crucial para garantir que a internet funcione de maneira eficiente.

Em uma arquitetura cliente/servidor, o cliente é geralmente um navegador web que solicita informações e o servidor é a máquina que responde a essas solicitações. Quando você digita um URL no seu navegador (cliente), ele envia uma solicitação ao servidor que hospeda o site. O servidor então responde enviando os arquivos necessários para o navegador exibir o site.



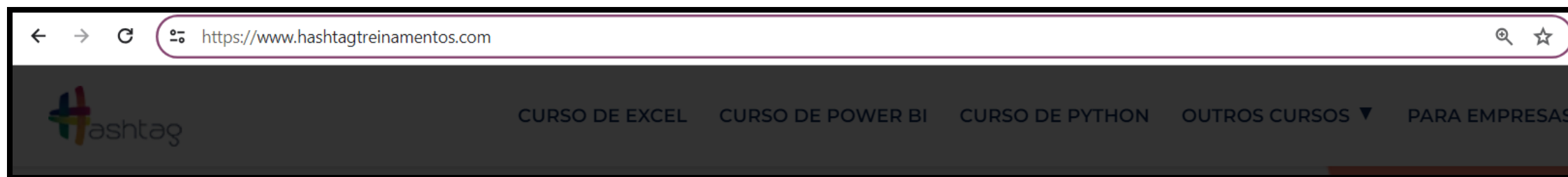
Primeiros Passos

HTTP ou HTTPS:

O HTTP, que significa Protocolo de Transferência de Hipertexto (em inglês, Hypertext Transfer Protocol), é o que torna a internet possível. É como um idioma que os computadores usam para se comunicar entre si quando você está navegando na web.

Pense no HTTP como uma conversa entre o seu navegador (como o Google Chrome, Mozilla Firefox, etc.) e um servidor web (onde os sites estão armazenados). Quando você digita um endereço da web e pressiona Enter, seu navegador envia um pedido (ou solicitação) para o servidor web pedindo para ver uma determinada página. Isso é chamado de solicitação HTTP.

O servidor web recebe o pedido, processa-o e envia de volta uma resposta para o seu navegador. Essa resposta contém as informações necessárias para exibir a página da web que você solicitou.



Primeiros Passos

- **Pedido (Request):** No contexto da internet, um pedido (request) é uma mensagem que seu navegador envia para um servidor web solicitando algum tipo de informação. Esse pedido pode ser para carregar uma página da web, um arquivo de imagem, um vídeo, ou até mesmo para enviar dados para o servidor, como ao preencher um formulário online.
- **Resposta (Response):** A resposta (response) é a mensagem que o servidor web envia de volta para o seu navegador após receber o pedido. Essa resposta contém a informação solicitada, como a página da web, o arquivo de imagem, ou os dados processados pelo servidor.

Então, o HTTP é como uma troca de mensagens entre seu navegador e o servidor web. Seu navegador faz pedidos e o servidor web responde com as informações necessárias para carregar e exibir as páginas da web que você deseja ver. É assim que você navega na internet todos os dias!

Agora vamos começar a programar o nosso arquivo `servidor.js`.

Primeiros Passos

Dentro do arquivo servidor.js, vamos utilizar algumas funcionalidades do Javascript.

```
const http = require("http");
```

Nessa linha de código em JavaScript, você está usando o **require** para importar o módulo **http**. Vamos entender o que isso significa:

- **require("http")**: O **require** é uma função do Node.js que permite importar módulos externos para o seu código. Neste caso, estamos importando o módulo **http**. O módulo **http** é um módulo integrado (built-in module) do Node.js que fornece funcionalidades para criar servidores HTTP e fazer solicitações HTTP.
- **const http**: Ao usar **const**, estamos declarando uma constante chamada **http** que armazenará o conteúdo do módulo importado. Isso significa que, uma vez que o módulo **http** é importado, não podemos atribuir outro valor a **http**.

Primeiros Passos

Uma outra funcionalidade que vamos utilizar é o Express que é um framework web para Node.js que simplifica o processo de criação de aplicativos da web e APIs (interfaces de programação de aplicativos). Ele fornece uma variedade de recursos poderosos para facilitar o desenvolvimento de servidores web de forma rápida e eficiente.

Você pode instalar o Express usando o npm (Node Package Manager), que é o gerenciador de pacotes do Node.js. Execute o seguinte comando no terminal:

```
npm install express
```

Depois de instalado, você pode importar e usar o Express em seus arquivos JavaScript.

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

Quando você instala dependências em um projeto Node.js usando o npm (Node Package Manager), uma pasta chamada **node_modules** é criada automaticamente no diretório do seu projeto. Esta pasta é onde o npm armazena todas as dependências (ou pacotes) que você instalou para o seu projeto.

Dentro da pasta **node_modules**, você encontrará subpastas para cada uma das dependências instaladas, junto com seus próprios arquivos e pastas. O npm gerencia todas essas dependências para você, garantindo que as versões corretas sejam instaladas e que todas as dependências necessárias para o seu projeto estejam disponíveis.

Primeiros Passos

As dependências são listadas no arquivo **package.json** sob duas seções principais:

Dependencies (Dependências): Esta seção lista as bibliotecas e módulos que são necessários para o funcionamento do seu aplicativo em um ambiente de produção. Isso inclui as dependências que seu aplicativo precisa para ser executado corretamente em um servidor ou em um ambiente de produção.

DevDependencies (Dependências de Desenvolvimento): Esta seção lista as bibliotecas e módulos que são necessários apenas para o desenvolvimento do seu aplicativo. Isso inclui ferramentas de teste, bibliotecas de compilação, plugins de minificação, entre outras ferramentas que ajudam no desenvolvimento, mas não são necessárias para a execução do aplicativo em um ambiente de produção.

O **package.json** é usado pelo npm (Node Package Manager) para instalar e gerenciar todas as dependências do seu projeto. Quando você executa o comando **npm install**, o npm lê o **package.json** e instala todas as dependências listadas no arquivo, garantindo que todas as bibliotecas e módulos necessários estejam disponíveis para o seu projeto.

```
{} package.json > ...
1  {
2    "name": "chat-treino-copia",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    > Debug
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "express": "^4.18.2"
15   }
16 }
```

Primeiros Passos

Nesse momento, com as dependências instaladas e o Express importado, vamos utilizar dentro de uma variável chamada "app".

```
const app = express();
```

A linha de código **const app = express();** cria uma instância do framework Express em seu aplicativo Node.js.

express() é uma função fornecida pelo framework Express. Quando chamada, ela retorna uma nova instância do Express. Esta instância representa o seu aplicativo Express e é usada para configurar rotas, adicionar middleware, lidar com solicitações HTTP e muito mais.

Então até o momento nosso arquivo servidor.js possui esta estrutura:

```
JS servidor.js > ...  
1  const http = require("http");  
2  const express = require("express");  
3  const app = express();
```

Parte 5

Criando a lógica do Servidor



Criando a lógica do Servidor

Agora vamos criar um servidor HTTP usando o módulo **http** do Node.js e associando a instância do Express (**app**) a esse servidor.

```
5  const servidorHttp = http.createServer(app);
```

- O **http** é um módulo integrado do Node.js que fornece funcionalidades para criar servidores HTTP.
- **createServer(app)**: **createServer** é um método do módulo **http** que cria um servidor HTTP. Ele recebe uma função como argumento, e essa função será chamada toda vez que o servidor receber uma solicitação HTTP. Neste caso, estamos passando o objeto **app**, que é a instância do Express que criamos anteriormente, como argumento para **createServer**. Isso significa que o Express será responsável por lidar com todas as solicitações HTTP recebidas pelo servidor.

A próxima instrução será para o servidor HTTP a escutar as solicitações que chegam na porta 3000.

```
7  servidorHttp.listen(3000);
```

Criando a lógica do Servidor

- **listen(3000): listen()** é um método do servidor HTTP que inicia o servidor e o faz escutar (ouvir) em uma porta específica. No exemplo, a porta específica é 3000.

Portanto, quando você chama **servidorHttp.listen(3000);**, você está dizendo ao servidor HTTP para começar a ouvir as solicitações que chegam na porta 3000 do computador onde o servidor está sendo executado.

Imagine que a porta 3000 é como uma porta de entrada para o seu aplicativo. Quando alguém faz uma solicitação para o seu aplicativo (por exemplo, digitando <http://localhost:3000> no navegador), o servidor HTTP estará ouvindo na porta 3000 e responderá a essa solicitação de acordo com as configurações que você definiu no seu aplicativo.

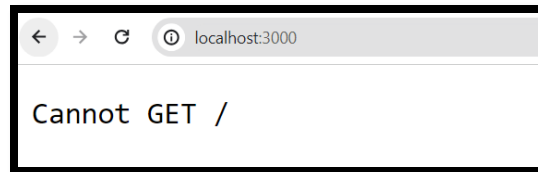
Para colocar o servidor para rodar, você precisa abrir o terminal e digitar o comando:

```
node ./servidor.js
```

Quando você executa esse comando no terminal e pressiona Enter, o Node.js irá iniciar a execução do arquivo **servidor.js**. Isso significa que o código contido nesse arquivo será executado, e se você configurou corretamente seu servidor dentro desse arquivo, ele começará a ouvir por solicitações HTTP e responderá de acordo com as configurações que você definiu.

Criando a lógica do Servidor

Com o servidor no ar, podemos acessar o endereço localhost:3000 e verificarmos que ele está no ar, porém está retornando uma mensagem de um erro.



A mensagem "Cannot GET /" é um erro comum que ocorre em aplicativos da web quando o servidor recebe uma solicitação para uma rota (ou URL) que não foi definida ou tratada pelo aplicativo.

Vamos quebrar a mensagem em partes:

- **"Cannot GET"**: Isso indica que o servidor recebeu uma solicitação HTTP para uma rota específica (nesse caso, a raiz do site, representada por "/"), mas não encontrou uma rota correspondente definida no aplicativo.
- **"/"**: O caractere "/" é frequentemente usado como a rota raiz de um site ou aplicativo da web. Representa o ponto de entrada principal do aplicativo.

Quando você vê a mensagem "Cannot GET /", geralmente significa que você está tentando acessar a raiz do seu aplicativo, mas o servidor não tem uma rota definida para lidar com essa solicitação específica. Para resolver esse erro, você precisa garantir que tenha definido uma rota correspondente para lidar com solicitações para a raiz do seu aplicativo. Isso é feito usando o Express (ou outro framework web) para configurar rotas em seu aplicativo.

Criando a lógica do Servidor

Quando você vê uma mensagem de erro "Cannot GET /" em seu aplicativo web, isso geralmente ocorre porque o Express não conseguiu encontrar uma rota correspondente para lidar com a solicitação feita pelo cliente. Isso pode acontecer especialmente quando o cliente está tentando acessar recursos estáticos, como arquivos HTML, CSS, JavaScript ou imagens.

Para corrigir esse erro e garantir que o Express possa lidar com solicitações para recursos estáticos de forma adequada, você pode utilizar o middleware **express.static()**.

```
JS servidor.js > ...
1  const http = require("http");
2  const express = require("express");
3  const app = express();
4
5  const servidorHttp = http.createServer(app);
6
7  app.use(express.static("public"));
8
9  servidorHttp.listen(3000);
10
```

Criando a lógica do Servidor

Ao usar **`app.use(express.static("public"))`**;, você está instruindo o Express a servir arquivos estáticos que estão localizados na pasta "public". Isso significa que quando o Express recebe uma solicitação para um recurso estático, ele procura esse recurso na pasta "public" e o envia de volta como resposta à solicitação.

Portanto, ao adicionar esse middleware, você está permitindo que o Express lide com solicitações para recursos estáticos de forma eficiente e evita que ocorram erros "Cannot GET /" quando os clientes tentam acessar esses recursos em seu aplicativo web.

Sempre que você fizer alterações no arquivo do servidor em um aplicativo Node.js, certifique-se de parar o servidor atual (se estiver em execução) como o comando **CTRL+C** e iniciar o servidor novamente usando o comando **node** para aplicar as alterações feitas. Isso garantirá que seu servidor esteja usando o código mais recente e atualizado.

```
node ./servidor.js
```

Criando a lógica do Servidor

Quando você faz alterações no arquivo do servidor em um aplicativo Node.js, geralmente precisa reiniciar o servidor para que as alterações entrem em vigor. Aqui está o porquê:

Node.js é um ambiente de tempo de execução assíncrono: O Node.js executa seu código de forma assíncrona, o que significa que ele carrega o código uma vez e o mantém em memória enquanto estiver em execução. Isso significa que, mesmo que você faça alterações no arquivo do servidor, o Node.js não reconhecerá automaticamente essas alterações e as aplicará ao servidor em execução.

Necessidade de reinicializar o servidor: Para aplicar as alterações feitas no arquivo do servidor, você precisa reiniciar o servidor Node.js. Isso significa interromper a execução do servidor (ou matar o processo) e iniciá-lo novamente. Ao fazer isso, o Node.js carregará o arquivo do servidor atualizado, aplicando as alterações feitas.

Processo de desenvolvimento: Em um ambiente de desenvolvimento, é comum fazer alterações frequentes no código do servidor para testar novos recursos, corrigir bugs ou fazer ajustes. Portanto, reiniciar o servidor após fazer alterações no arquivo do servidor é uma prática comum e necessária para garantir que as alterações sejam refletidas no servidor em execução.

Criando a lógica do Servidor

Para evitar o incômodo de reiniciar nosso servidor a cada mudança, vamos usar o seguinte comando: **node --watch ./servidor.js**. O **--watch** é uma ferramenta que fica de olho em arquivos específicos e, sempre que eles mudam, reinicia automaticamente o servidor Node.js. Isso é super útil enquanto estamos desenvolvendo, porque nos permite ver as alterações aplicadas no servidor sem ter que reiniciá-lo manualmente toda vez.

Um novo conceito que vamos aprender é Websocket / Socket:

Socket:

Um socket é uma abstração de comunicação bidirecional entre processos. Ele permite que programas em diferentes computadores se comuniquem diretamente uns com os outros. Os sockets são usados para estabelecer uma conexão de rede entre um cliente e um servidor. Eles podem ser TCP (Transmission Control Protocol) ou UDP (User Datagram Protocol), com TCP sendo mais comum para comunicação bidirecional confiável.

- Imagine um "cabo" virtual que conecta dois computadores pela internet.
- Esse "cabo" permite que eles troquem informações diretamente, como mensagens ou arquivos.
- Os sockets são como esses "cabos", permitindo a comunicação direta entre computadores.

Criando a lógica do Servidor

WebSocket:

WebSocket é um protocolo de comunicação bidirecional em tempo real sobre uma única conexão TCP. Ele fornece uma maneira de estabelecer uma conexão persistente entre um cliente e um servidor. Ao contrário do HTTP, que é baseado em solicitação e resposta, o WebSocket permite que o servidor envie dados para o cliente sem que o cliente precise solicitar. Isso permite a construção de aplicativos da web em tempo real, como chats online, jogos multiplayer, atualizações em tempo real e assim por diante.

- O WebSocket é como um tipo especial de "cabo" virtual, mas usado especialmente para sites.
- Ele permite que um site envie informações para o navegador do usuário sem precisar esperar por uma solicitação.
- É útil para aplicativos que precisam de atualizações em tempo real, como bate-papos ou jogos online.

Resumindo, o Socket é a ideia geral de comunicação direta entre computadores, enquanto o WebSocket é uma tecnologia específica para sites que precisam de comunicação em tempo real. Ambos são essenciais para diferentes tipos de comunicação online.

Criando a lógica do Servidor

Para utilizarmos o Socket, vamos encerrar nosso servidor com atalho CTRL+C e executar o comando "npm install socket.io".

O comando **npm install socket.io** é utilizado para instalar o pacote **socket.io** em um projeto Node.js. Aqui está uma explicação mais detalhada:

socket.io é uma biblioteca JavaScript para aplicativos web em tempo real. Ela facilita a comunicação bidirecional em tempo real entre clientes e servidores da web. Com **socket.io**, você pode criar aplicativos da web que enviam e recebem dados instantaneamente, permitindo interações em tempo real, como bate-papos, colaboração em tempo real e atualizações de conteúdo dinâmico.

Portanto, quando você executa **npm install socket.io**, o NPM baixa e instala o pacote **socket.io** e suas dependências no seu projeto Node.js, permitindo que você comece a usar a funcionalidade de comunicação em tempo real oferecida pelo **socket.io**.

```
PS C:\Users\mikag\OneDrive\Documentos\Projetos\Chat-Treino-Copia> npm install socket.io
[#####] \ reify:socket.io: http fetch GET 200 https://registry.npmjs.org/socket.io/-/socket.io-4.7.4.tgz 1790ms (cache miss)
```

Criando a lógica do Servidor

Vamos armazenar agora a funcionalidade do Socket dentro de uma variável, assim como fizemos com o Express.

```
4  const funcionalidadesIO = require("socket.io");  
5  
6  const servidorHttp = http.createServer(app);  
7  const io = funcionalidadesIO(servidorHttp);
```

Vamos analisar essas linhas de código:

const funcionalidadesIO = require("socket.io");

- Aqui, estamos importando o módulo **socket.io** para o nosso projeto.
- O **require("socket.io")** retorna uma função que podemos usar para configurar e interagir com o Socket.IO em nosso servidor.

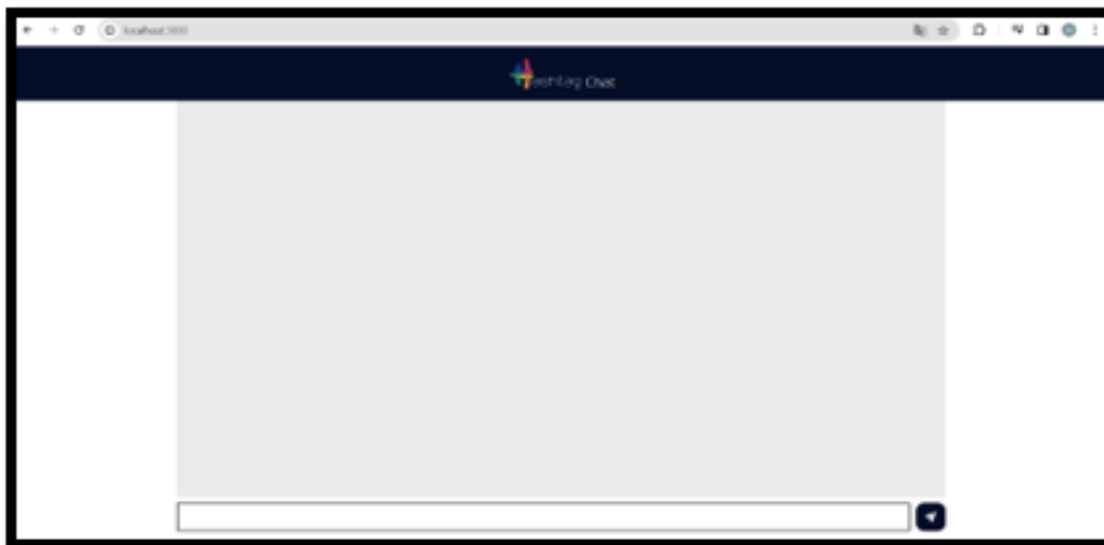
const servidorHttp = http.createServer(app);

- Aqui, estamos criando um servidor HTTP utilizando o módulo **http** do Node.js.
- O método **http.createServer(app)** cria um servidor HTTP que irá lidar com as requisições HTTP feitas ao nosso aplicativo Express, onde **app** é uma instância do Express.

Criando a lógica do Servidor

const io = funcionalidadesIO(servidorHttp);:

- Aqui, estamos inicializando o Socket.IO no nosso servidor HTTP.
- Passamos o objeto **servidorHttp** que criamos anteriormente para o **funcionalidadesIO**, que é a função retornada quando importamos o módulo **socket.io**.
- Isso conecta o Socket.IO ao nosso servidor HTTP, permitindo que ele escute por eventos de WebSocket e gerencie comunicação em tempo real entre o servidor e os clientes conectados.



Parte 6

Criando a lógica da conexão Servidor x Front-end



Criando a lógica da conexão Servidor X Front-end

Vamos adicionar dentro da pasta "public" um arquivo chamado "interface.js". E escrever o seguinte código dentro dele:

```
public > JS interface.js > ...  
1   const socket = io();
```

Dentro do arquivo index.html do projeto, iremos corrigir a importação do arquivo que acabamos de criar:

```
29   <script src="/socket.io/socket.io.js"></script>  
30   <script src="./interface.js"></script>
```

Vamos voltar ao nosso arquivo servidor.js, e vamos configura-lo para "escutar" um evento do Socket.io.

```
11   io.addListener("connection");
```

Criando a lógica da conexão Servidor X Front-end

A linha de código **io.addListener("connection")** está configurando um listener para o evento "connection" no Socket.IO. Vamos analisar mais detalhadamente:

io: A variável **io** representa o objeto principal do Socket.IO. Ela é utilizada para interagir com a biblioteca Socket.IO e gerenciar conexões e eventos em tempo real.

.addListener("connection"): O método **addListener()** (ou seu equivalente **.on()**) é usado para registrar um callback para um determinado evento. Neste caso, estamos registrando um callback para o evento "connection". Esse evento é emitido quando um cliente se conecta ao servidor através do Socket.IO.

Quando um cliente se conecta ao servidor, o evento "connection" é disparado, e qualquer código dentro do callback associado a esse evento será executado. Isso permite que o servidor realize ações específicas quando novos clientes se conectam, como inicializar comunicação, autenticar usuários, ou configurar canais de comunicação.

Geralmente, dentro do callback do evento "connection", é onde são definidas as ações e lógica relacionadas à interação com o cliente recém-conectado, como responder a mensagens, transmitir dados para outros clientes ou realizar operações de inicialização.

Criando a lógica da conexão Servidor X Front-end

Agora vamos criar a nossa função callback, que servirá de instrução após o evento "connection".

```
11 function alguemEntrouNaSala() {  
12 |   console.log("alguém entrou na sala");  
13 | }  
14  
15 io.addListener("connection", alguemEntrouNaSala);
```

Vamos executar novamente nosso servidor:

```
Restarting './servidor.js'  
PS C:\Users\mikag\OneDrive\Documentos\Projetos\Chat-Treino-Copia> node --watch ./servidor.js  
(node:13392) ExperimentalWarning: Watch mode is an experimental feature and might change at any time  
(Use `node --trace-warnings ...` to show where the warning was created)  
alguém entrou na sala
```

Criando a lógica da conexão Servidor X Front-end

Queremos que nossa função vá além de apenas notificar quando alguém entra na sala; queremos implementar mais algumas funcionalidades.

Para isso, vamos criar uma nova função que será usada como callback para o nosso evento "connection", e vamos comentar a que acabamos de criar.

```
JS servidor.js > ...
1  const http = require("http");
2  const express = require("express");
3  const app = express();
4  const funcionalidadesIO = require("socket.io");
5
6  const servidorHttp = http.createServer(app);
7  const io = funcionalidadesIO(servidorHttp);
8
9  app.use(express.static("public"));
10
11  /* function alguemEntrouNaSala() {
12     console.log("alguém entrou na sala");
13  } */
14
15  function comportamentoDoSocket(socket) {
16     console.log("novo usuário na sala");
17     socket.addListener("nova mensagem", (msg) => {
18         io.emit("nova mensagem", msg);
19     });
20  }
21
22  io.addListener("connection", comportamentoDoSocket);
23
24  servidorHttp.listen(3000);
25  |
```

PROBLEMS OUTPUT TERMINAL ... node + - [] [] ... ^

```
Restarting './servidor.js'
novo usuário na sala
novo usuário na sala
```

Criando a lógica da conexão Servidor X Front-end

```
function comportamentoDoSocket(socket) {  
  console.log("novo usuário na sala"); // Mostra uma mensagem quando um novo usuário se conecta à sala  
  
  // Adiciona um listener para o evento "nova mensagem"  
  socket.addListener("nova mensagem", (msg) => {  
    // Quando uma nova mensagem é recebida, ela é transmitida para todos os clientes conectados  
    io.emit("nova mensagem", msg);  
  });  
}
```

- A função **comportamentoDoSocket** é chamada sempre que um novo usuário se conecta à sala.
- Ela exibe uma mensagem no console informando que um novo usuário entrou na sala.
- Em seguida, ela adiciona um listener para o evento "nova mensagem" associado ao socket do usuário recém-conectado.
- Quando uma nova mensagem é recebida nesse socket, a função transmite essa mensagem para todos os clientes conectados à sala usando **io.emit()**.
- Isso permite que todas as mensagens enviadas por qualquer usuário sejam distribuídas para todos os outros usuários na sala em tempo real.

Criando a lógica da conexão Servidor X Front-end

Com o comportamento criado, é importante lembrar que na comunicação pela internet, um lado está enviando informações (o back-end) e o outro lado está recebendo essas informações (o front-end). Até agora, só configuramos o lado que envia (o back-end), mas não implementamos nada no lado que recebe (o front-end). Portanto, precisamos configurar essa conexão no nosso arquivo **interface.js**.

```
3   const chat = document.getElementById("mensagens"); // <div id="mensagens"></div>
4
5   ✓ socket.addEventListener("nova mensagem", (msg) => {
6   ✓   // Aqui, estamos adicionando um listener para o evento "nova mensagem" do Socket.IO.
7     // Quando uma nova mensagem é recebida do servidor, este código será executado.
8     // A mensagem recebida é armazenada na variável "msg".
9     // Aqui você pode adicionar a lógica para exibir essa mensagem na interface do usuário.
10  });
```

Este trecho de código define um listener para o evento "nova mensagem" do Socket.IO. Quando uma nova mensagem é enviada do servidor para o cliente, o código dentro da função de callback será executado. Você pode adicionar lógica dentro desse callback para manipular a mensagem recebida, como exibi-la na interface do usuário.

Criando a lógica da conexão Servidor X Front-end

Armazenamos dentro de uma variável um elemento div de dentro do nosso index.html.

```
12 <div id="mensagens"></div>
```

Dentro desse elemento, vamos criar elementos de listas `` para cada mensagem escrita no nosso chat já com a estilização implementada do nosso arquivo CSS.

```
4 socket.addEventListener("nova mensagem", (msg) => {  
5   const mensagem = document.createElement("li"); // <li></li>  
6   mensagem.textContent = msg; // <li>Mensagem do Alon</li>  
7   mensagem.classList.add("mensagem"); // <li class="mensagem">Mensagem do Alon</li>  
8 });
```

Este trecho de código cria um novo elemento `` no DOM (Documento Objeto Modelo) do navegador usando JavaScript. Aqui está o que cada linha faz:

1. **`const mensagem = document.createElement("li");`**; Cria um novo elemento HTML `` e o armazena na variável **`mensagem`**.
2. **`mensagem.textContent = msg;`**; Define o conteúdo de texto do elemento `` criado anteriormente com o conteúdo recebido na variável **`msg`**. Isso adiciona o texto da mensagem ao elemento ``.
3. **`mensagem.classList.add("mensagem");`**; Adiciona a classe CSS "mensagem" ao elemento ``. Isso permite que você aplique estilos específicos a todas as mensagens usando CSS.

Criando a lógica da conexão Servidor X Front-end

Agora vamos adicionar a mensagem (representada pelo elemento **** criado anteriormente) ao final da lista de mensagens no chat, garantindo que ela seja exibida para o usuário.

```
public > JS interface.js > ...
1  const chat = document.getElementById("mensagens"); // <div id="mensagens"></div>
2  const socket = io();
3
4  socket.addEventListener("nova mensagem", (msg) => {
5    const mensagem = document.createElement("li"); // <li></li>
6    mensagem.textContent = msg; // <li>Mensagem do Alon</li>
7    mensagem.classList.add("mensagem"); // <li class="mensagem">Mensagem do Alon</li>
8    chat.appendChild(mensagem); // <div id="mensagens"><li class="mensagem">Mensagem do Alon</li></div>
9  });
```

appendChild(mensagem): Este método JavaScript é usado para adicionar um nó filho ao elemento específico. No caso, estamos adicionando o elemento **mensagem** como um novo item de lista (****) ao elemento **chat**.

Criando a lógica da conexão Servidor X Front-end

Com a criação dos novos elementos de mensagens, queremos permitir que o usuário escreva um texto que seja interpretado pelo JavaScript. Depois que o usuário pressionar o botão "Enviar", queremos que o JavaScript consiga enviar esse texto para o chat.

Então precisamos armazenar os dois elementos que são responsáveis pela caixa de entrada de texto e o botão dentro do arquivo interface.js, para isso vamos procurar o id deles, dentro do arquivo index.html:

```
14 <input id="texto" autocomplete="off" autofocus />  
15 <button id="enviar">
```

Index.html

Interface.js

```
2 const botaoEnviar = document.getElementById("enviar");  
3 const elementoTexto = document.getElementById("texto");
```

Feito isso, vamos adicionar um evento ao nosso botão para que ele seja capaz de escutar o evento de clique e implementarmos a instrução que queremos que a função callback execute.

```
botaoEnviar.addEventListener("click", () => {});
```

Criando a lógica da conexão Servidor X Front-end

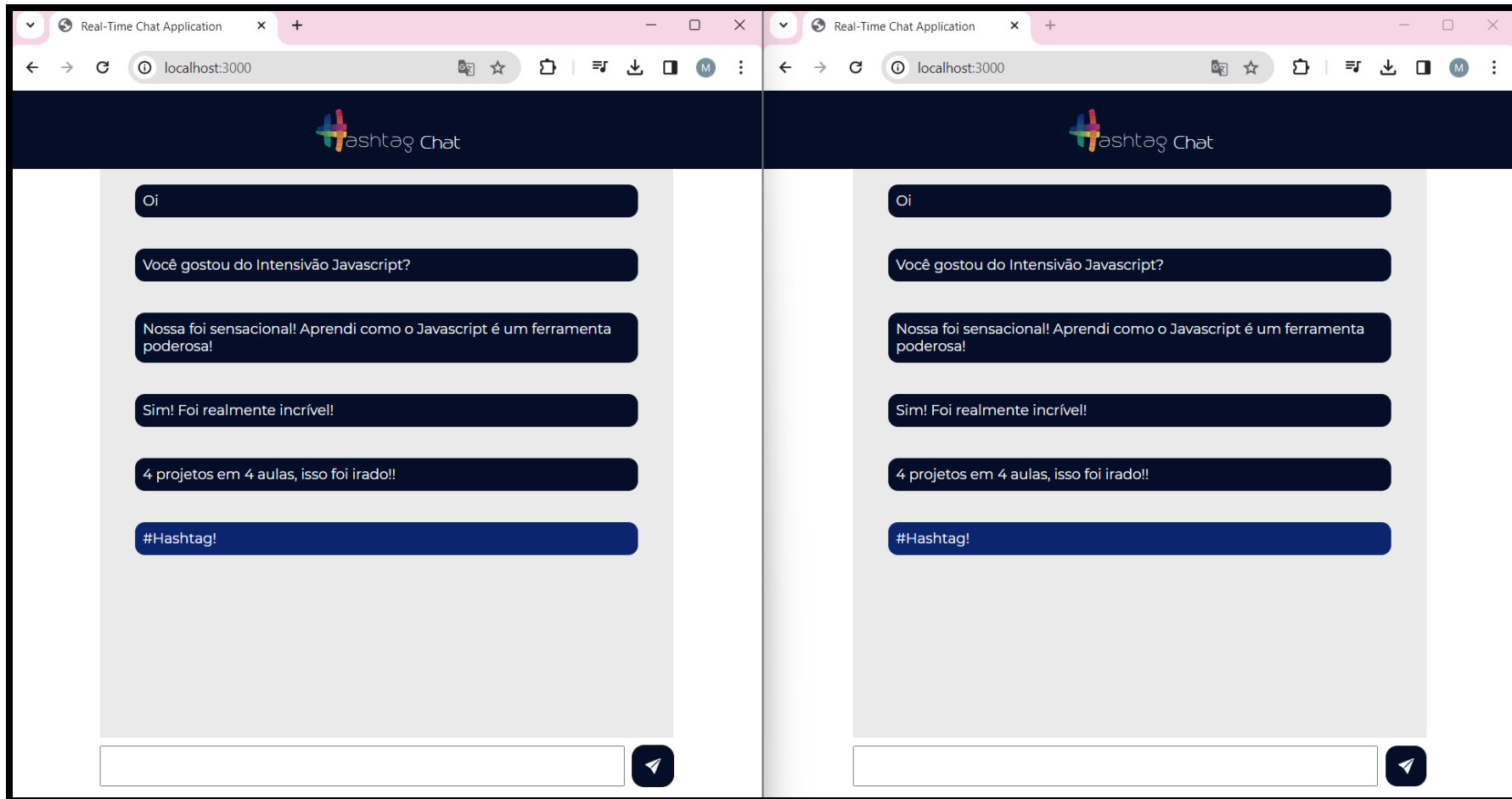
```
botaoEnviar.addEventListener("click", () => {  
  const mensagemAEnviar = elementoTexto.value;  
  socket.emit("nova mensagem", mensagemAEnviar);  
  elementoTexto.value = "";  
});
```

- **botaoEnviar.addEventListener("click", () => { ... })**: Este trecho de código adiciona um ouvinte de evento ao botão "Enviar". Quando o botão é clicado, a função dentro das chaves { ... } será executada.
- **const mensagemAEnviar = elementoTexto.value**;; Aqui, estamos capturando o valor digitado pelo usuário no campo de texto (representado por **elementoTexto**) e armazenando-o na variável **mensagemAEnviar**.
- **socket.emit("nova mensagem", mensagemAEnviar)**;; Utilizamos o objeto **socket** para emitir (enviar) um evento chamado "nova mensagem" para o servidor. O segundo argumento da função **emit()** é a mensagem que queremos enviar para o servidor.
- **elementoTexto.value = ""**;; Após enviar a mensagem, limpamos o campo de texto, definindo seu valor como uma string vazia, garantindo que o campo fique pronto para receber uma nova mensagem.

Resumindo, quando o usuário clica no botão "Enviar", o texto digitado é enviado para o servidor através do Socket.IO, e em seguida o campo de texto é limpo para permitir que o usuário digite uma nova mensagem.

Criando a lógica da conexão Servidor X Front-end

Pronto! Nosso Chat em Tempo Real está rodando, nosso servidor está conectado com a interação do usuário!



Ainda não segue a gente no Instagram e nem é inscrito no nosso canal do Youtube? Então corre lá!



@hashtagtreinamentos



youtube.com/hashtag-treinamentos

