



Instituto Federal do Triângulo Mineiro
Análise e Desenvolvimento de Sistemas

Redes de Computadores
Documentação do Trabalho Prático

Patrocínio - MG
2025
ALEXSSANDER JOSÉ DE OLIVEIRA DE CASTRO
LUCAS AMARAL LUCIANO
PABLO VINÍCIUS LIMA SOUZA

Redes de Computadores
Documentação do Trabalho Prático

Envio de documento em PDF com o objetivo de documentar um Sistema com aplicação de API da Binance no curso de Análise e Desenvolvimento de Sistemas

Prof: Júnio.

1.Introdução.

Neste trabalho, você encontrará a documentação relacionada ao desenvolvimento de um site voltado para a exibição de informações sobre criptomoedas, utilizando dados fornecidos pela API oficial da Binance. O objetivo principal deste projeto foi praticar o consumo de uma API pública, integrando-a a uma aplicação web capaz de apresentar valores e gráficos atualizados de diferentes ativos do mercado de criptoativos. Através do uso da API da Binance, foi possível acessar dados em tempo real, interpretar respostas no formato JSON e transformá-las em visualizações claras e dinâmicas para o usuário final. Este documento tem como função registrar as principais etapas do processo de desenvolvimento, desde a integração com a API até a estruturação da interface do site

2.Criação do servidor Backend.

Criar um servidor backend é importante quando usamos uma API como a da Binance porque ajuda a manter o sistema mais seguro e organizado. Com o backend, dá pra esconder informações sensíveis, como chaves de acesso, que não deveriam ficar visíveis no navegador. Além disso, ele serve pra filtrar e preparar os dados antes de mandar pro site, e também ajuda a controlar a quantidade de requisições feitas, evitando problemas com limites da API. Outro ponto importante é que, em muitos casos, o próprio navegador bloqueia requisições feitas diretamente do frontend por questões de segurança, como as regras de CORS. O backend resolve esse problema servindo como intermediário autorizado entre o navegador e a API.

2.1 - Implementação do Server.js (SERVIDOR BACKEND):

```

graphic > src > serverjs > ...
1  const express = require('express');
2  const port = 3001;
3
4  const app = express();
5
6  const cors = require('cors');
7  app.use(cors());
8
9  const axios = require('axios');
10
11 app.get('/klines', async (req, res, next) => {
12   const { symbol, interval } = req.query;
13   if(!symbol || !interval) return res.status(422).send('Symbol and interval are required');
14
15   try{
16     const response = await axios.get(`https://api.binance.com/api/v3/klines?symbol=${symbol}&interval=${interval}&limit=60`);
17     res.json(response.data);
18   }catch(err){
19     res.status(500).json(err.response ? err.response.data : err.message);
20   }
21
22 })
23
24 app.listen(port);
25 console.log('server listening...');

```

Esse código cria um servidor backend usando Node.js com o framework Express, com o objetivo de buscar dados da API pública da Binance e repassá-los para o frontend. Ele roda na porta 3001 e permite que o navegador acesse seus dados ativando o CORS. Ao acessar a rota /klines, o servidor espera receber dois parâmetros pela URL: symbol (símbolo da criptomoeda) e interval (intervalo de tempo dos candles). Se esses parâmetros não forem informados, o servidor responde com erro. Caso os parâmetros estejam corretos, ele usa a biblioteca Axios para fazer uma requisição à API da Binance e busca os últimos 60 candles da criptomoeda informada. Os dados retornados são então enviados diretamente como resposta para o frontend. Se houver algum erro na requisição, o servidor responde com uma mensagem de erro e o status 500. Esse backend funciona como intermediário entre o navegador e a API da Binance, evitando problemas com CORS, escondendo detalhes técnicos da API e organizando melhor a estrutura da aplicação.

2.2 Requisição dos dados

Nos focamos no gráfico de velas, então na requisição de dados para facilitar a implementação do mesmo dividimos os dados em estruturas chamadas “Candle”, vela em inglês, e cada vela possui tempo de abertura (openTime), valor de abertura (open), valor mais alto (high), valor mais baixo (low), e valor de fechamento (close).

```

1  export default class Candle{
2    constructor(openTime, open, high, low, close){
3      this.x = new Date(openTime);
4      this.y = [parseFloat(open),parseFloat(high),parseFloat(low),parseFloat(close)];
5    }
6  }

```

E então fazemos a requisição, consumimos a API utilizando o Axios e criamos um array de Candles que é retornado com todas as velas necessárias para o gráfico:

```

1 import axios from 'axios';
2 import Candle from './Candle';
3
4 export async function getCandles(symbol, interval){
5     const response= await axios.get(`http://localhost:3001/klines?symbol=${symbol.toUpperCase()}&interval=${interval}`);
6     const candles = response.data.map(k =>{
7         return new Candle(k[0], k[1], k[2], k[3], k[4]);
8     })
9     return candles;
10 }

```

2.3 Criação do gráfico

Este componente React cria um gráfico de velas (candlestick) utilizando a biblioteca react-apexcharts, que serve como wrapper para o ApexCharts. O gráfico é renderizado de forma dinâmica, recebendo os dados por meio da propriedade props.data, o que permite reutilização e atualização automática conforme os dados mudam. A configuração do gráfico é feita através do objeto options, onde o eixo X (xaxis) é definido como do tipo datetime e formatado para exibir as datas no padrão brasileiro (dd/MM HH:mm), ajustado para o fuso horário de São Paulo. Isso é feito usando um formatter que converte os timestamps para um formato legível com toLocaleString.

O eixo Y (yaxis) possui tooltips ativados para facilitar a leitura dos valores ao passar o mouse sobre as velas. Os dados são organizados dentro da series, com cada ponto contendo um objeto com x (data/hora) e y (um array com os valores de abertura, máxima, mínima e fechamento). Por fim, o gráfico é renderizado com tipo candlestick, largura de 800 pixels e altura de 600 pixels

```

1  import ApexCharts from 'react-apexcharts';
2
3  export default function Chart(props){
4
5      const options = {
6          xaxis: {
7              type: 'datetime',
8              labels: {
9                  datetimeUTC: false,
10                 format: 'dd/MM HH:mm',
11                 formatter: function (value, timestamp) {
12                     return new Date(timestamp).toLocaleString('pt-BR', {
13                         timeZone: 'America/Sao_Paulo',
14                         hour: '2-digit',
15                         minute: '2-digit',
16                         day: '2-digit',
17                         month: '2-digit',
18                     });
19                 },
20             },
21         },
22         yaxis:{
23             tooltip: {
24                 enable: true
25             }
26         }
27     }
28
29     const series =[{
30         data: props.data
31     }]
32
33     return(
34         <ApexCharts options={options}
35             series={series}
36             type="candlestick"
37             width={800}
38             height={600}/>
39     )
40 }

```

3. Implementação do gráfico no Front

O código começa importando os hooks `useEffect` e `useState` do React, além do componente `Chart`, a função `getCandles`, o hook `useWebSocket` da biblioteca `react-use-websocket`, e a classe `Candle`. Em seguida, o componente `App` é definido com dois estados: `data`, que armazena os candles a serem exibidos no gráfico, e `moeda`, que representa o par de criptomoeda e moeda selecionadas (inicialmente 'BTCBRL').

Dentro do `useEffect`, que roda apenas uma vez ao carregar o componente, o código lê a moeda a partir dos parâmetros da URL (por exemplo: `?moeda=ETHBRL`) e

atualiza o estado moeda. Logo após, ele chama a função `getCandles`, que busca os candles históricos de 1 minuto da moeda escolhida, e armazena esses dados no estado `data`. Se ocorrer algum erro, ele exibe um alerta com a mensagem.

```
1 import {useEffect, useState } from 'react';
2 import Chart from './Chart';
3 import { getCandles } from './DataService';
4 import useWebSocket from 'react-use-websocket';
5 import Candle from './Candle';
6
7 function App() {
8   const [data, setData] = useState([]);
9   const [moeda, setMoeda] = useState('BTCBRL');
10
11   useEffect(() => {
12
13     const urlParams = new URLSearchParams(window.location.search);
14     const moedaSelecionada = urlParams.get('moeda');
15     setMoeda(moedaSelecionada);
16
17     getCandles(moedaSelecionada, '1m')
18       .then(data => setData(data))
19       .catch(err => alert(err.response ? err.response.data : err.message));
20   }, []);
```

Na sequência, é estabelecida uma conexão WebSocket com a API da Binance, usando o par da moeda em minúsculas e o intervalo de 1 minuto. Ao abrir a conexão, é exibida uma mensagem no console. A configuração permite reconectar automaticamente a cada 3 segundos em caso de falha. O `onMessage` é executado sempre que uma nova mensagem chega, representando um novo candle.

Dentro do `onMessage`, o código cria um novo objeto `Candle` com os dados da mensagem (abertura, fechamento, máxima, mínima, etc.). Ele copia os dados atuais (`data`) para um novo array `newData`. Se o candle ainda estiver se formando (`k.x === false`), ele substitui o último candle do array. Se o candle estiver fechado, remove o primeiro da lista e adiciona o novo no final, mantendo o gráfico sempre atualizado com os últimos candles. Por fim, atualiza o estado `data`.

O return do componente exibe um título com o nome da moeda e renderiza o componente `Chart`, passando o estado `data` como propriedade para desenhar o gráfico.

```

22 const {lastJsonMessage} = useWebSocket(`wss://stream.binance.com:9443/ws/${moeda.toLowerCase()}@kline_${'1m'}`,{
23   onOpen: () => console.log('Connected to Binance'),
24   onError: (err) => console.log(err),
25   shouldReconnect: () => true,
26   reconnectInterval: 3000,
27   onMessage: ()=>{
28     if(lastJsonMessage){
29       const newCandle = new Candle(lastJsonMessage.k.t,lastJsonMessage.k.o,lastJsonMessage.k.h,lastJsonMessage.k.l,lastJsonMessage.k.c);
30       const newData = [...data];
31       if(lastJsonMessage.k.x === false){
32         newData[newData.length - 1] = newCandle;
33       }
34       else{
35         newData.splice(0, 1);
36         newData.push(newCandle);
37       }
38       setData(newData);
39     }
40   }
41 },
42 ))
43
44 return (
45   <div>
46     <h2>Gráfico: {moeda}</h2>
47     <Chart data={data} />
48   </div>
49 );
50 }
51
52 export default App;

```

3.1 Preço médio atualizado mostrado na tela:

Para mostrar o preço e o atualizar em tempo real fizemos 4 funções de acordo com as Criptomoedas a serem mostradas e atualizar o texto em HTML:

```

graphic > src > callGraphicjs > ...
1 window.addEventListener('DOMContentLoaded', () => {
2   setInterval(() => {
3     fetch("https://api.binance.com/api/v3/avgPrice?symbol=BTCBRL")
4     .then(response => response.json())
5     .then(data => {
6       const preco = data.price;
7       const elementoPreco = document.getElementById('preco-btc');
8       if (elementoPreco) {
9         elementoPreco.textContent = `R$ ${Number(preco).toLocaleString('pt-BR', { minimumFractionDigits: 2 })}`;
10      }
11    })
12    .catch(error => console.error("Erro ao buscar preço:", error));
13  }, 1000);
14 });

```

O código começa registrando um ouvinte de evento com `window.addEventListener('DOMContentLoaded', ...)`, que garante que o JavaScript só será executado depois que todo o HTML tiver sido carregado. Dentro desse evento, é criado um `setInterval` que executa uma função a cada 1000 milissegundos (ou seja, **a cada 1 segundo**). Essa função faz uma requisição HTTP com `fetch` para a API pública da Binance, no endpoint <https://api.binance.com/api/v3/avgPrice?symbol=BTCBRL>. Esse endpoint retorna o preço médio atual do Bitcoin em reais (BRL).

Depois da resposta chegar, ela é convertida em JSON com `.json()`, e então o valor do campo `price` é extraído e guardado na constante `preco`.

Em seguida, o script tenta localizar no HTML um elemento com o ID `preco-btc` usando `document.getElementById`. Se esse elemento existir, o código atualiza seu conteúdo

de texto (textContent) com o preço formatado em reais (R\$) com duas casas decimais, usando toLocaleString com a localidade pt-BR.

3.2 Resultado na tela

E por ultimo apenas utilizamos um Iframe para poder exibir o gráfico na tela que implementamos e estilizamos.

