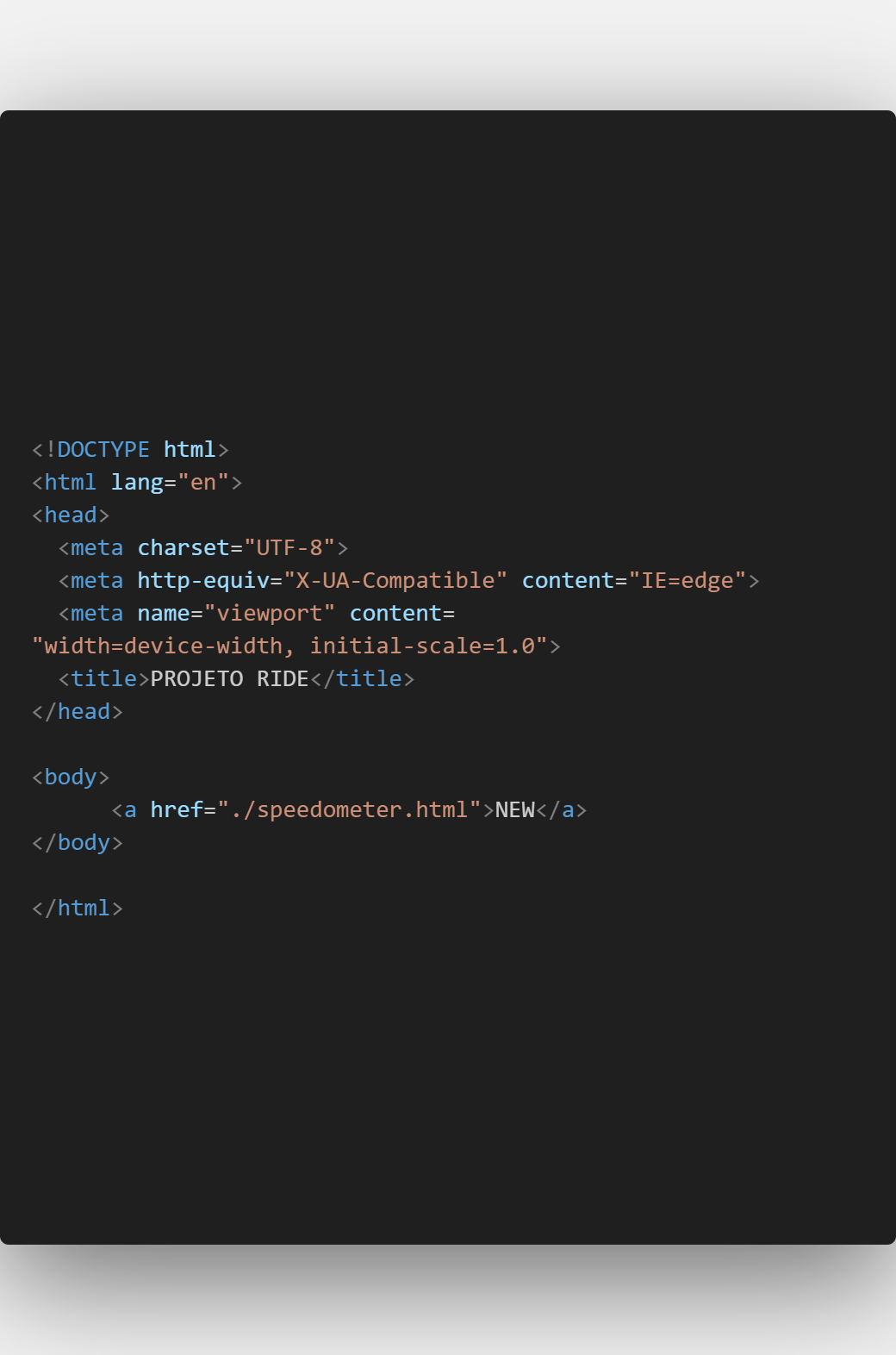
**PROJETO RIDE**

**Criando a tela de velocímetro e o estilo dela**

Neste projeto, criamos 3 arquivos html, o index, o speedometer e o detail. Cada um desempenhando uma função de utilização para o usuário. A index com uma lista das corridas e características das ações do usuário, a speedometer com o velocímetro ligado a um gps e a detail com analises mais aprofundados das atividades.

**Tela do index provisória:**



Neste começo, apenas montamos um link para a tela do speedometer.

**Começo da tela de speedometer:**



Neste projeto estamos utilizando bootstrap e alguns estilos próprios.

Para isso o link do bootstrap obrigatoriamente deve vir antes do link do nosso arquivo css.

Para que possamos preencher 100% da tela, precisamos usar a classe “h-100” no <html> e no <body>.

OBS1: A tag <nav> serve para definirmos um grupo específico de links. Alêm disso, ajuda os navegadores a identificarem na hora das buscas.

Obs2: A classe “btn btn-close” do bootstrap, monta um botão em formato de X para ser usado no fechamento de algo.

Obs3: A classe “flex-fill” faz com que o elemento preenche todo o espaço referente ao elemento pai. Se houver mais de um elemento com “flex-fill” eles terão sua larguras iguais dentro do espaço.

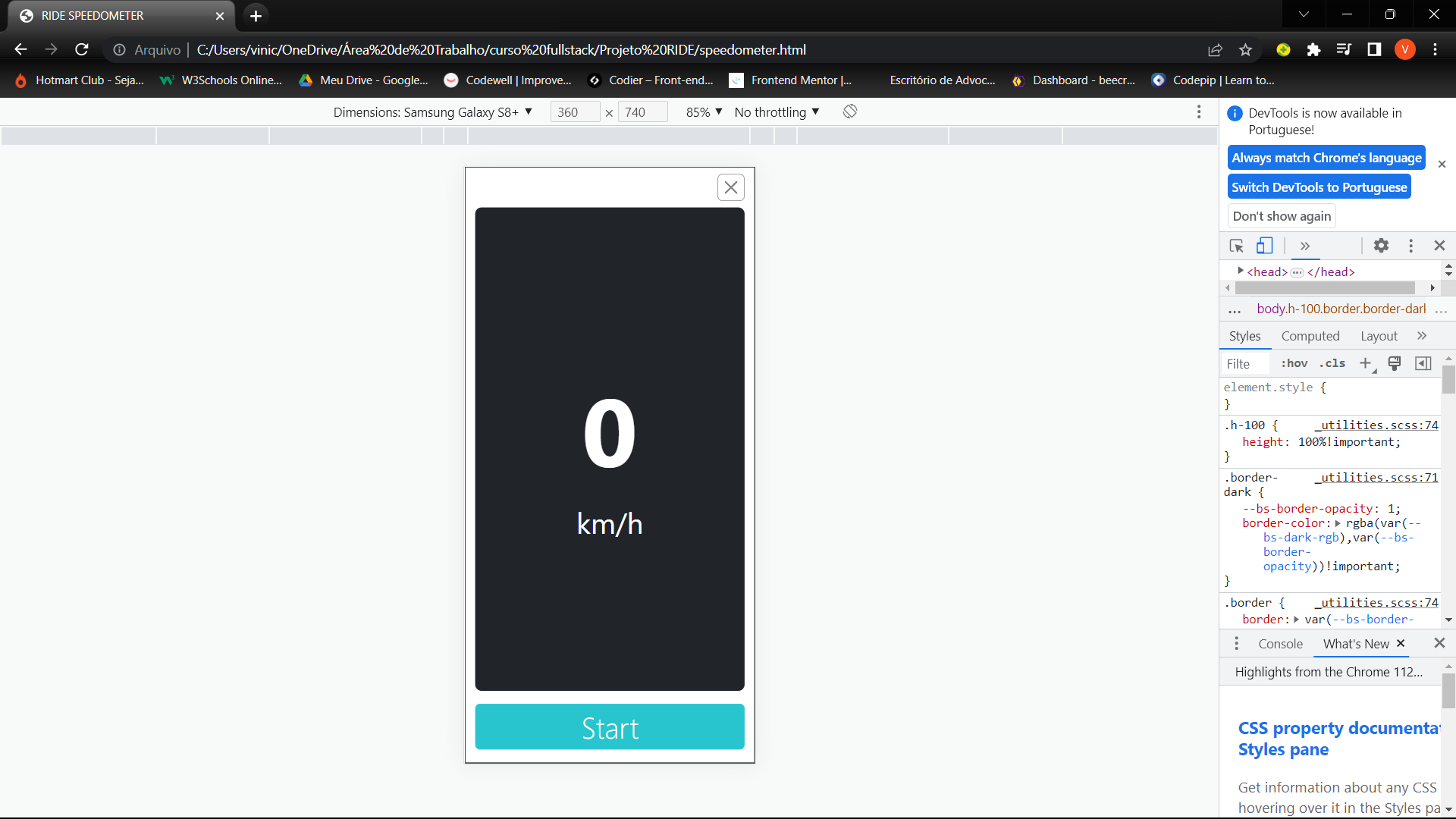
Obs4: A classe” fw-bold” (font weight) configura o texto em negrito.

Obs5: Em alguns casos, podemos escolher usar nossos próprios estilos. Para isso, podemos inserir um <style> in line, interno ou externo no código, ou podemos alterar as próprias variações do bootstrap em um arquivo css externo.



Alterando algum elemento dentro do :root, toda vez que essa classe for usada ela terá a configuração que montamos.

Neste arquivo css, copiamos as classes do bootstrap dentro do nosso console para que possamos alterar alguma classe ou elemento específico.



**API E COMO ACESSAR O GPS DO CELULAR**

API significa **Aplication Programn Interface,** e na prática é uma interface entre aplicações. Então podemos usar uma api para acessar a aplicação de gps do celular e ligá-la a nossa aplicação.

Nesta aplicação vamos utilizar a API navigator: geolocation.

Porêm, para conseguirmos utilizar esta API, precisamos possuir uma conexão segura, E neste primeiro momento, nosso aplicativo não possui um acesso https, já que ela está no nosso local storage.

Para reverter isso, usamos a aplicação **ngrok**, que cria um link seguro dentro de uma conexão https enquanto nossa aplicação está em desenvolvimento. Isto permite por exemplo, que compartilhemos nosso projeto com outra pessoa em outro país.

Para usar o ngrok, precisamos nos cadastrar no site ngrok.com. Após o cadastro, o site entrega um token de segurança que deverá ser utilizado dentro do terminal de comando. Feito isso, devemos ativar o live server do editor de texto e copiar o número da porta gerado, em seguida, usamos o comando “ngrok http” e o número de porta gerado dentro do terminal de comando. Isso criará um link que pode ser compartilhado, permitindo assim que outros dispositivos entrem na aplicação.

**CAPTURANDO OS DADOS DE LOCALIZAÇÃO**



Para que a contagem possa ser encerrada, e a numeração seja zerada, precisamos usar a propriedade clearWath na variável watchID, que recebeu anteriormente o método navigator. Em seguida watchID fica igual a nulo novamente.

Criamos a variável “watchID” para permitir a aplicação permanecer zerada antes do botão start e depois do botão stop.

Aqui, o ponto de interrogação representa um “if”, porém, de forma ternária.

O navigator é o objeto que vai expor a API de geolocalização através da comunicação com o gps do aparelho

Esta parte permite que os botões de start e stop troquem de lugar conforme o click do usuário.

Neste início, criamos 3 variáveis “const”, por que esses elementos serão usados por todo o código daqui por diante. E usamos o “querySelector” para buscar os seletores correspondentes aos elementos que precisamos alterar.

Obs: A propriedade “classList” permite adicionar métodos a classes. Removendo, adicionando, alterando de maneira mais simples e ágil.

Obs: Criamos um arquivo “speedometer.js” para criar a funcionalidade de geolocalização.

Obs: A propriedade “watchPosition” retorna um valor referente a posição do usuário. Recebe 3 atributos: successCallBack(que é executada sempre que recebe um novo dado), errorCallBack(que é executada quando houver algum erro), e o objeto Options, que fornece configurações adicionais a medição de posição. Como por exemplo o nível de precisão da medição. Como visto em:

 const options = {enableHighAccuracy : true}

Obs: Nesta parte:

speedElement.innerText = position.coords.speed ? (position.coords.speed \* 3.6).toFixed(1) : 0

Precisamos montar uma condição em que, caso a velocidade seja nula, o valor retornado seja “0”. Então usamos o if ternário (?) para dizer que, caso a velocidade seja nula, o valor retornado será 0.

Obs: Ainda se tratando da última observação, a medida de velocidade utilizada pelo método “speed” é metros/segundo. Sendo assim, multiplicamos por 3,6 para transforma-la em kiliometros/hora, e adicionamos a propriedade toFixed() para dizer que queremos apenas uma casa decimal.

Obs: Na função startBtn; utilizamos o:

 if(watchID)

    return

Para definirmos que, se o watchID não for nulo, todo o processo se inicializará.

E na função stopBtn, usamos o:

    if(!watchID)

    return

Caso ele seja nulo, o contador permanecerá zerado.

**ARMAZENANDO OS DADOS NO LOCAL STORAGE**

Nesta etapa, fizemos a aplicação registrar os dados do percurso do usuário no **local stotage**. Para isso criamos um segundo arquivo “.js” chamado “storage.js”. Onde as funções usadas no speedometer.js ficaram descritas.

**Speedometer.js**

****

Logo depois de apertarmos o botão stop, não podemos usar nenhum outro dado. Para isso, acrescentamos um “currentRide = null” depois do “upsateStopTime”.

Criamos o “updateStopTime” para que possamos gravar o fim da corrida, assim teremos o inicio da corrida com o “createNewRide e o fim com o “updateStopTime.””

Criamos esta função para que possamos adicionar novas posições no “currentRide”.

E criamos esta função, que fará o registro de novas corridas. Por isso é importante ela estar descrita antes do “watchPosition”, porque senão o “wacthPosition” pode receber dados antes da função de registro ser executada, causando um erro.

Criamos esta variável que começa nula

**Storage.js**



Aqui, nós acrescentamos as informações de posição ao array do “rideRecord”. Usando os valores da propriedade “coords” e “timeStamp”. Para isso, usamos o método “push”.

Ver obs.

Ver obs.

Aqui armazenamos nossos dados no local storage, onde a chave será o “rideID” e o valor retornado será o “rideRecord”

Aqui criamos outra constante. Mas agora como um objeto, que terá um array de dados como posicionamento, velocidade, etc. E também a marcação do momento de início e de fim da corrida.

Nesta função, criamos uma constante que dará um identificador para cada corrida, usando o” Date.now()”

Obs: No speedometer.html, o storage.js vem antes do speedometer.js.

   <script src="./storage.js"></script>

   <script src="./speedometer.js"></script>

Porque usaremos coisas do storage dentro do speedometer, mas não o contrário.

Obs: O método “Date.now()” do Javascript retorna a quantidade de milissegundos desde 01/01/1970. Está é a data que o relógio interno do Javascript começa a contar o tempo.

Porém, no nosso caso, podemos usa-la para criar um ID para cada corrida diferente, através do momento em que o usuário aperta o botão START.

Obs:

function getRideRecord(rideID){

    return JSON.parse(localStorage.getItem(rideID))

}

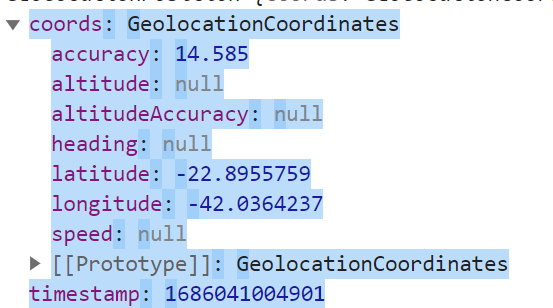
function addPosition(rideID, position){

    const rideRecord = getRideRecord(rideID)

O que está acontecendo aqui é o seguinte, estamos adicionando uma nova posição no “handleSuccess” através do “addPosition”. Para isso criamos a função auxiliar “getRideRecord(rideID)” que retornará os dados do “localStorage” referente ao “rideID” correspondente. Porém, este dado retorna como uma string, então usamos um JSON.parse para transforma-lo em um objeto.

Obs: O método **push()**, acrescenta um elemento ao final do array.

Obs: Os valores do **coords** copiamos do próprio console.



Obs: Agora que acrescentamos a “position” no “rideRecords” através do “rideID”, é preciso salva-lo novamente. Para isso, criamos uma nova função chamada “saveRideRecords”. E a usamos dentro do “addposition”

function saveRideRecord(rideID, rideRecord){

    localStorage.setItem(rideID, JSON.stringify(rideRecord))

 rideRecord.data.push(newData)

    saveRideRecord(rideID, rideRecord)

Obs:

function updateStopTime(rideID) {

    const rideRecord = getRideRecord(rideID)

    rideRecord.stopTime = Date.now()

    saveRideRecord(rideID, rideRecord)

}

Nesta função, acrescentamos ao “rideRecord” o momento exato em que o usuário aperta o botão stop. Para isso usamos o **Date.now** no “stopTime” já existente. Em seguida sal vamos ele com o “saveRideRecords” como visto anteriormemnte.

**CRIANDO A LISTA DE REGISTROS**

Para começar, criamos dentro do index.html a tag <li> e demos a ela um “id”

 <ul id="rideList">

   </ul>

Feito isso, precisamos trazer os dados coletados pelo app para abastecer essa lista. Então criamos outro arquivo javascript, chamado **index.js**.

No index.js, criamos uma constante para pegar o id “rideList”, que criamos no index.html, em seguida criamos uma função para coletar os dados de todas as corridas feitas.

const rideListElement = document.querySelector("#rideList")

const allRides = getAllRides()

Obs: todas as funções desse projeto estão concentradas no arquivo storage.js. Isso ajuda a limpar o código evitando ter de repetir a mesma função em todos os arquivos que ela se repete. E também facilita quando estamos usando um arquivamento de dado local ou em nuvem.

Obs: Para que funcione é preciso posicionar a tag script do storage.js antes da tag script do index.js no arquivo index.html. Porque o index está usando uma função do storage.

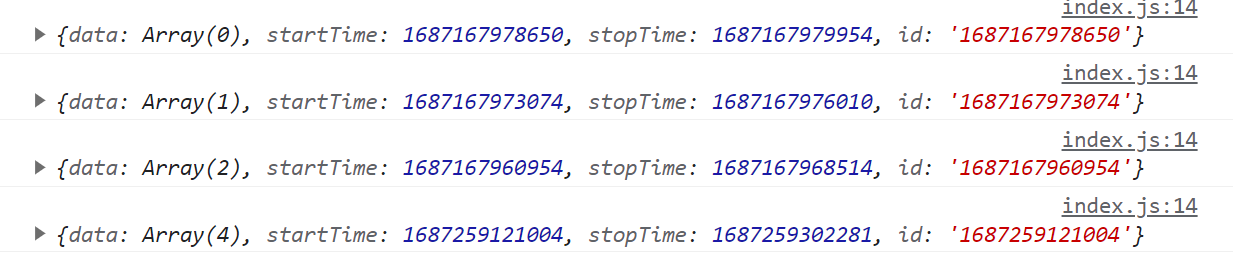
O primeiro passo na função “getAlLRides” é transforma os dados das corridas de objeto para array. Para isso, usamos um método da classe object chamado “entries”.

Obs: O método “entries()” retorna um array objeto com seu parâmetro e valor.

function getAllRides(){

    return Object.entries(localStorage)

}



Voltando ao index.js usaremos agora um forEach porque já transformamos os dados em um array objeto.



Criamos esta nova constante ride para que possamos transformar os dados das corridas do “value” de array para objeto. Permitindo também que possamos atribuir ids para cada corrida.

O método “appendChild” acrescenta um novo elemento ao final da lista de elementos relecionadas.

Usamos o “createElement” para criar um novo elemtento “li”

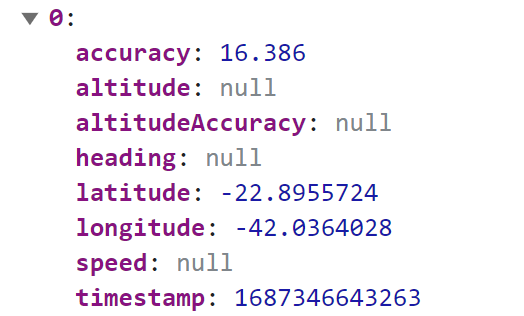
Um array é formado por um parâmetro e um valor, por isso, os parâmetros para esta função de seta são o id e o value

Obs: método forEach(), acrescenta uma função para cada elemento de um array. Como resultado, ele percorrerá toda a coleção disponibilizando a chave e o valor de todos os itens.

**COLOCANDO DADOS NA LISTA PARTE 1**

Neste momento, iremos inserir dados na nossa lista de armazenamento. São eles: local, velocidade máxima e distância percorrida.

Começaremos com o local. Para isso usaremos os dados de latitude e longitude da posição **[0]** do array que conseguimos anteriormente com o elemento **“ride”.**



Então, para usarmos a primeira posição criamos uma const para definir isso:

const firstPosition = ride.data[0]

Porém, não é possível convertermos estes números em um local de forma direta. Para isso usaremos uma API, que fará uma chamada na web, para retornar cidade, estado, país, etc.

Criamos uma função para isso, e criamos uma const com a url da API.

function getLocationData(latitude, longitude){

    const url = `https://api.bigdatacloud.net/data/reverse-geocode-client

No entanto, a url em si, não retorna o valor que queremos sozinha. Precisamos definir os parâmetros que serão usados. Para fazermos isso, acrescentamos um ponto de interrogação no final da url. Isso defini que a partir daquele momento, tudo será parâmetro e língua local para serem usados pela API.

function getLocationData(latitude, longitude){

    const url = `https://api.bigdatacloud.net/data/reverse-geocode-client?latitude=${latitude}&longitude=${longitude}&=localitylanguage=en`

Para podermos pegar os dados da nossa url e inseri-los no código, usamos o método **fetch**.

Obs: O método fetch() captura dados de alguma fonte/servidor online, retornando uma **promise** para algum objeto de resposta.

Para facilitar o código, não usamos a resposta dentro do **then()** que é natural do **fetch()**. Usamos o **await** para jogarmos esta resposta para dentro da constante que criamos.

Mas se usamos o **await**, precisamos transformar a nossa função em uma função **async**.

Para finalizar a função, o que queremos não é a constante **response**, o que queremos é o **json** dessa constante.

Obs: o método **json()** é um formato de transporte armazenamento de dados normalmente usado quando existem dados sendo enviados de um servidor para uma página web.

O **json()** também retorna uma **promise**, então usaremos novamente o **await**

Esta função completa ficou assim:



Agora podemos por esta função para funcionar. Vamos usar nossa nova função dentro do **forEach()**. Mas é preciso lembrar que, uma função **async** retorna uma outra promise, então precisamos novamente usar o **await** e o **async** para corrigir isso.

allRides.forEach(async ([id,value])=>{

    const ride = JSON.parse(value)

    ride.id = id

    const firstPosition = ride.data[0]

    const firstLocationData = await getLocationData(firstPosition.latitude,firstPosition.longitude)

Para esse projeto, usaremos como informação de localização a cidade e o código do país. Estes dois dados nós vamos buscar dentro da nossa função **getLocationData**.

Então, criamos uma nova constante chamada **FirstLocationsData**, e adicionamos nossa função. Feito isso, no intuito de continuar adicionando dados de informação para o usuário, criaremos elementos “Divs” no nosso código.

Com esse objetivo, criamos uma nova constante chamada **cityDiv** e usamos o método **document.createElement()**. Em seguida adicionamos com **innerText** uma **string literals** com os dados que buscamos (city e country code).

E para podermos adicionar constantemente novas “Divs” com novos elementos usamos o método **appendChild()** na nossa constante **itemElement**.

const cityDiv = document.createElement("div")

    cityDiv.innerText = `${firstLocationData.city} - ${firstLocationData.countryCode}`

itemElement.appendChild(cityDiv)

Obs: O uso do “**LITERALS**” permite que possamos interpolar variáveis e expressões em strings de uma maneira fácil, usando a sintaxe **`${}`** .

O próximo dado que iremos inserir no aplicativo, será a velocidade máxima alcançada na corrida. Para isso, criamos uma nova função que pegará todas as posições registradas, chamada **getMaxSpeed(positions){}**, e que também será um **array**.



Multiplicamos por 3.6 para transformar a medição de m/s para km/h. E usamos o método **toFixed()** para definir quantas casas decimais usaremos depois da vírgula.

Aqui, aplicamos um **forEach**, para verificar se cada posição que existe é maior do que **0** ou se é **nula**. Então, se **positions.speed** for diferente de **0** e não for **nulo**, a variável **maxSpeed** vai receber o **positions.maxSpeed**.

Primeiro criamos uma variável para informar que nossa velocidade começa zerada

E para inserirmos estes dados na tela do usuário, usaremos a mesma lógica vista antes para informar o local.

 const maxSpeedDiv = document.createElement("div")

     maxSpeedDiv.innerText = `Max Speed : ${getMaxSpeed(ride.data)} km/h`

itemElement.appendChild(maxSpeedDiv)

Próximo passo é obter a distância percorrida pelo usuário. Então criamos outra função que também usará todas as posições, chamada **getDistance(positions)**.

Nela, criamos uma constante para definir o raio de circunferência da Terra em km, que é 6.371. Em seguida uma variável para definir a distância total que começa com 0.

Usamos também um **for()** para conseguirmos pegar a quantidade de posições e também, para definirmos que precisamos de pelo menos 2 pontos de distância para alcançar o resultados esperado(neste caso, 2 pontos de latitude e 2 de longitude).

Até aqui o **for()** ficou desse jeito:



É executada sempre antes da próxima avaliação do loop. Normalmente, atualiza ou incrementa a variável do contador

Define a condição para a execução do bloco

Normalmente inicia a contagem de variáveis, e é executada uma vez antes da execução do bloco.

O **for()** executa um bloco de comando várias vezes (LOOP). Ele é formado por 3 expressões opcionais.

For( inicialização, condição, expressão final)

Obs: Na nossa expressão de condição, definimos que é preciso que a variável i deve ser menor que as quantidades de positions, porém, para conseguirmos o resultado, o i não pode ser zero. Como o positions.lenght já é 1, usamos o -1 para definir que se não houver pelo menos 1 posição, o código não irá executar.

Criamos agora, outras duas constantes capturando as posições mencionadas anteriormente.



Aqui, definimos o **[i + 1]** para pegarmos a próxima posição logo depois da primeira capturada pela **const p1**. Isso funciona, porque definimos que o **i** deve ser menor do que a posição no **array -1**. Pois se ele for igual, ele não vai capturar nenhuma posição, buscando assim sempre a penúltima posição dentro do **array**.

Continuando dentro do **for()**, Criamos uma nova constante para definirmos a diferença do p2 e do p1, a fim de, descobrir a posição exata do usuário. No entanto, estas posições de latitude e longitude são representadas em “graus”, e para o nosso código precisamos que elas estejam em “radianos”.

Criamos então, uma função que possa transforma-las de uma medida para outra. Então a nova constante e a nova função ficam desta forma:

const deltaLatitude = toRad(p2.latitude - p1.latitude)

        const deltaLongitude = toRad(p2.longitude - p1.longitude)

function toRad(degree){

        return degree \* Math.PI / 180

    }

Obs: Sempre veremos o **PI** quando estivermos tratando de círculos.

A partir de agora, faremos um cálculo envolvendo seno, cosseno, raiz quadrada, arco tangente da diferença dos elementos p1 e p2 da circunferência, multiplicação, soma, etc.

Não é necessário aprender a matemática por trás do código. Quando se deparar com algum problema do parecido, é preciso buscar se alguém já resolveu antes.



E aqui, usamos o **totalDistance** que se iniciava com **0** e acrescentamos o **distance** que definimos agora.

Aqui, conseguimos calcular a distância entre o p1 e o p2

No final da função **getDistance(positions)**, retornamos o **totalDistance** com o método **toFixed(1)** para definirmos apenas uma casa decimal após a vírgula.

return totalDistance.toFixed(1)

E agora, acrescentamos a distância percorrida a nossa tela, como fizemos anteriormente.

const distanceDiv = document.createElement("div")

     distanceDiv.innerText = `Distance : ${getDistance(ride.data)} km`

itemElement.appendChild(distanceDiv)

O próximo item que precisamos acrescentar é a duração da corrida.

Começamos criando uma nova constante e um novo appenchild como visto anteriormente.

const durationDiv = document.createElement("div")

    durationDiv.innerText = getDuration(ride)

 itemElement.appendChild(durationDiv)

Obs: Usamos novamente o ride como argumento da função, porque nele, nós já definimos o momento que começamos a corrida e o momento em que encerramos ela em milissegundos. Então a nova função se baseará no stopTime – startTime.

**A função getDuration(ride)** fica assim:



Criamos uma nova função para ajustarmos o formato da hora como desejamos que apareça para o usuário. Definindo que seja um numero e a quantidade de dígitos.

Como baseamos nosso tempo em milissegundos, precisamos dividir nosso calculo por 1000

Obs: O método **padString()** preenche a string original com um determinado caractere, conjunto de caracteres ou quantidade de caracteres. Até que atinja o comprimento fornecido.

Sintaxe:

str.padStart(targetLength [, padString])

targetLength

O comprimento da string resultante uma vez que a string alvo tenha sido preenchida. Caso seu valor seja menor do que o comprimento da string alvo, é retornado a própria string alvo, sem alterações.

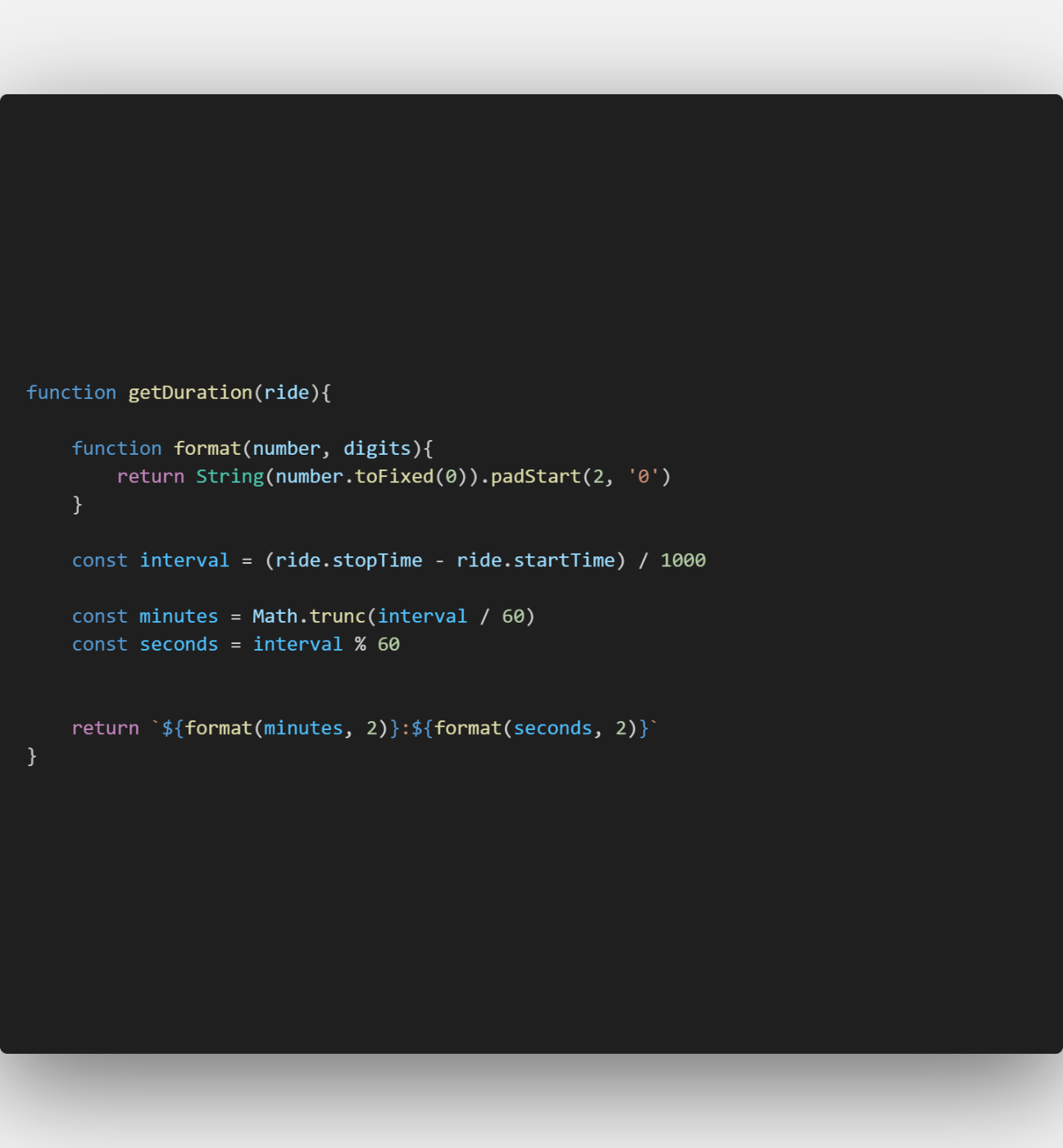
padString

Opcional. O caractere, ou caracteres, que deve preencher a string alvo. Caso o comprimento dessa essa string de preenchimento seja superior ao targetLength, ela será truncada a partir da direita. O valor padrão é " "

Para finalizarmos a duração da corrida, ajustamos o formato dos minutos e segundos do percurso.

Individualmente, criamos a constante **minutes** que vai pegar a outra constante **interval** e dividi-la por 60. E criamos também a constante **seconds,** que será 60% do **interval** para podermos pegar os decimais da divisão do **interval** por 60.

A função **getDuration(ride){}** ficou assim:



Usamos o **Math.trunc** para retornar a parte inteira do número.

Retornamos a função **format** da maneira que definimos (number, digits).

A última informação que queremos é da data do evento. Novamente criamos nossa **div e appendChild**.

 const dateDiv = document.createElement("div")

    dateDiv.innerText = getStartDate(ride)

 itemElement.appendChild(dateDiv)

Criamos então nossa nova função chamada **getStartDate(ride)**, que também usa o **ride** como argumento.



Criando constantes para cada elemento que queremos que apareça no app, conseguimos formatar a data como quisermos

Usamos o **hour12 : false** porque não queremos que apareça o AM ou PM.

O formato **long** só pode ser usado para elementos que possuam “nome”. Como mês e dia da semana.

Obs: o **new Date()** constrói datas como objetos. E por padrão, retorna uma string com a time zone do navegador do usuário.7

Exemplo:

**Sat Jul 08 2023 12:13:01 GMT-0300 (Horário Padrão de Brasília)**

Existem 9 maneiras de criar objetos com datas:

new Date(): usa a data atual do navegador  
new Date(date string): cria um objeto de data através de uma string de data.

A partir daqui, cria objetos com uma data e tempo específico  
  
new Date(year,month)  
new Date(year,month,day)  
new Date(year,month,day,hours)  
new Date(year,month,day,hours,minutes)  
new Date(year,month,day,hours,minutes,seconds)  
new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)

O Javascript conta os meses de 0 a 11. Um número superior a 11 irá transpor o ano e passar ao ano seguinte.

Obs: O método **toLocaleString()** retorna um objeto data como uma string, localizado para uma língua específica, usando as configurações do próprio método.

Alguns dos formatos permitidos são:

|  |  |
| --- | --- |
| dateStyle | Legal values: "full" "long" "medium" "short" |
| timeStyle | "full" "long" "medium" "short" |
| localeMatcher | "best-fit" (default) "lookup" |
| timeZone |  |
| hour12 | false true |
| hourCycle | "h11" "h12" "h23" "h24" |
| formatMatcher | "basic" "best-fit" (default) |
| weekday | "long" "short" "narrow" |
| year | "2-digit" "numeric" |
| month | "2-digit" "long" "narrow" "numeric" "short" |
| day | "2-digit" "numeric" |
| hour | "2-digit" "numeric" |
| minute | "2-digit" "numeric" |
| second | "2-digit" "numeric" |
| timeZoneName | "long" "short" |

O index.js completo ficou assim até esta etapa:



No nosso arquivo index.html, inserimos nosso link para o **bootstrap** e para nosso arquivo **global.css**, dentro do <head>

  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css" rel="stylesheet"

  integrity="sha384-KK94CHFLLe+nY2dmCWGMq91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJOZ" crossorigin="anonymous">

  <link href="./global.css" rel="stylesheet">

Nosso objetivo inicial é usar um ícone no lugar da palavra “NEW”, aproveitando que estamos usando o link **CDN** do **Bootstrap** usaremos o link **CDN** para ícones dentro do nosso **global.css**, dessa forma poderemos sempre usar esses ícones fazendo o link com o **global.css**.

**Global.css**

@import url("https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css");

Nosso botão de ícone ficou assim:



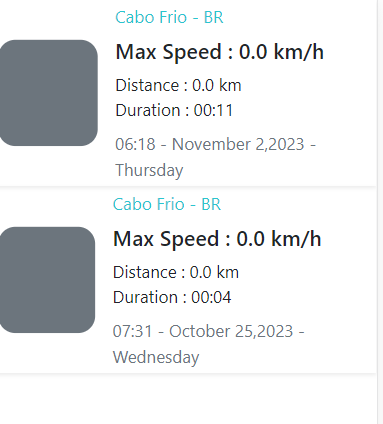
Este é o código para o ícone de “+” no Bootstrap;

Este estilo garante que nada sobreponha o botão.

O estilo dos itens da lista ficou assim:







Obs: O mapa ainda não está finalizado.

Resolvendo o bug de ordem dos itens da lista.



Este “rideListElement” estva no final dos “itemElement”, causando a desordem dos itens. Passando para antes do “await”, garantimos que todos os elementos serão carregados na ordem correta.

Este “await” estava antes do item, então, todo elemento esperava antes de ser carregado. Sendo que, alguns elementos esperam mais do que outros, causando o problema. Passando para esta posição eles aguardam serem carregados na ordem correta.

**Criando a tela de detalhe**

Criamos novos arquivos, detail.html, dataManager.js e detail.js.

Transferimos todas as funções do index.js para o dataManager.js, porque são as mesmas funções utilizadas na nova página, mas para a aplicação funcionar precisamos criar um acesso <script> a estas funções, então, no index.html acrescentamos uma tag <script>dataManager</script>.

Para acessar a página de detalhes, acrescentei no index.js um addEventListener:



O ? é um operador de condicional que neste caso está definindo o id do detail.html

Para que o index.js possa reconhecer o detail.html após o evento “click,” transformei o URL do detail.html em um parâmetro.

O objeto window.location pode ser usado para obter o endereço da página atual (URL) e redirecionar o navegador para uma nova página.

O uso do href retorna a URL da página deseja ou da página atual.

No Detail.js para encontrar o detail.html usei o seguinte método



A propriedade search retorna a parte da string de consulta de uma URL, incluindo o ponto de interrogação (?). A propriedade search também pode ser usada para definir a string de consulta.

Um desses métodos é o “get”, que retorna o primeiro valor associado ao parâmetro de pesquisa.

A interface URLsearchParams Define métodos utilitários para se trabalhar com os parâmetros de uma URL

Como os dados do detalhe são os mesmos da página de informações salvas das corridas, copiei os mesmos dados do index.js para o detail.js.

No entanto, para que o firstPosition com await funcione precisei criar uma função assíncrona para as constantes seguintes.



Somado a uma função assíncrona, o carregamento do conteúdo é acelerado.

O evento DOMContentLoaded é acionado quando todo o HTML foi completamente carregado e analisado, sem aguardar pelo CSS, imagens, e subframes para encerrar o carregamento.

Não é possível dar um append ao document, então, criei uma div no detail.html, com o id de “data”. Dessa forma poderei formatar os estilo do mapa mais facilmente.



Desta forma, eu consigo dar um append no #data e não ao document.

Para finalizar, copiei o x para sair da página lá do speedometer.html a classe “nav-bar” e coloquei no detail.html. Ajustei também o estilo da lista e a posição do mapa. Finalizei criando um botão de deletar na páginas de detalhes. Todos esses arquivos até este momento ficaram assim:



Detail.html

****

Detail.js

****

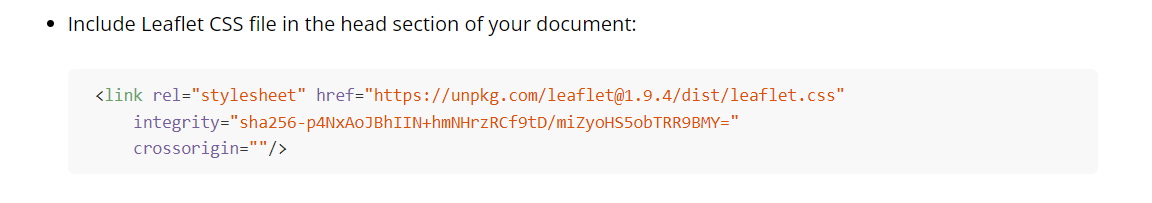
dataManager.js

**Mapa e percurso**

Neste momento coloquei um mapa na tela de detalhes. E para isso, usei uma api externa no site leaflet.com. Este site é um projeto open source que interage com Javascript e que permite que busquemos mapas para nosso próprio projeto.

No próprio site existem tutoriais de como usar a biblioteca de mapas, no meu caso apenas tive que copiar e colar as partes do código que me interessavam.

Este vai no head:



Este vai no final do body:



Uma vez que usamos o link para o mapa, ganhamos acesso ao objeto **L**, que é um objeto criado pelo site para manipular seus mapas.

Outra coisa que precisei foi escolher o tipo de mapas (tilemap) ideal para o meu projeto. Para isso usei as opções disponíveis do leaflet, no link:

<https://leaflet-extras.github.io/leaflet-providers/preview/>

Algumas das opções de mapas necessitam de uma api ou chave de acesso do próprio desenvolvedor, mas preferi escolher uma que não precisasse disso.



A parte grifada é o que copiei para o código.

No arquivo detail.js eu criei uma constante dentro do carregamento do document para configurar o mapa:



O addTo() é a função que permite que me permite adicionar o tilelayer a constante map

O tilelayer é o que copiamos no site para definir o tipo de mapa que que entrará no projeto. Todos os mapas possuem informaçãoes que podemos alterar no código, como zoom e atribuition.

A longitude e latitude já haviam sido definidas antes. Apenas copiei

Para o mapa funcionar precisamos do método setview, para mostrar as coordenadas iniciais do mapa. O setview possui um array com dois objetos (latitude e longitude) e um valor pra o zoom

Na função map, o que é buscado são identificadores, desta forma não é necessário o # para encontrar o id

Para criar o tracejado do percurso que o usuário percorreu, criei um array com um conjunto de pares de latitudes e longitudes que eu já havia definido antes no código.



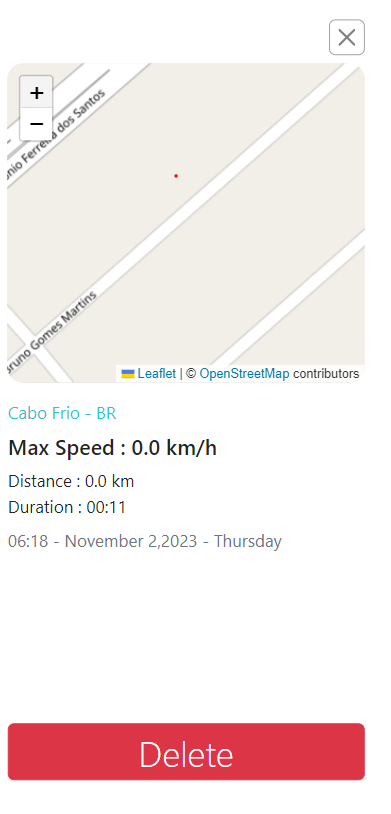
O método fitbounds e getbounds do map, permite que o mapa centralize no trajeto percorrido e ainda adicione um zoom para facilitar a visualização.

O método polyline do map nos permite adicionar uma linha de trajetória e definior o estilo dela

Nesta função, eu retorno a informação de latitude e longitude que o firstPosition do ride.data trouxe anteriormente e atribuo isso a uma constante.

Esta parte completa do código ficou assim:





**Botão de delete**

Para criar a função deletar, primeiro criei um identificador para a tag <button> no detail.html chamada “deleteBtn”, em seguida criei uma costante no detail.js para acrescentar uma função onde vamos pedir para que o localStorage apague os dados selecionados(esta função ficará armazenada no arquivo storage.js). Aproveitei e defini que toda vez que o botão delete e o botão stop forem acionados, a página volte para a de registros.

**Detail.js**



**Storage.js**



**Mapa na lista com marcador**

Nesta etapa, criei um mapa na página de lista, que até este momento estava apenas com o elemento criado.

Inicialmente, inseri o link css do leaflet no head do index.html e em seguida o link Javascript do leaflet no body do index.html.



Criei uma constante porque eu reutilizei este id em outro momento.

O elemento já estava criado, o que faltava era um ID para a div do mapElement. Sem um identificador, não conseguimos inserir o mapa no elemento.



Este elemento insere no mapa um marcador de posição. Como já havia definido o firstPosition, bastou copiar o código para esta posição.

Estes são alguns exemplos de alterações que o leaflet nos permite utilizar para configurar os mapas.

Eu reutilizei este trecho do detail.js. Apenas acrescentei alguns elementos.