Semana 2

Desenvolvimento web

HTTP

Hypertext Transfer Protocol: é um protocolo que permite a obtenção de recursos, como documentos HTML. É a base de qualquer troca de dados na Web e um protocolo cliente-servidor, o que significa que as requisições são iniciadas pelo destinatário, geralmente um navegador da Web. Um documento completo é reconstruído a partir dos diferentes sub-documentos obtidos, como por exemplo texto, descrição do layout, imagens, vídeos, scripts e muito mais.

HTTP

Existem dois tipos de mensagens, requisições e respostas, cada uma com seu próprio formato.

Requisições HTTP (request)

O protocolo HTTP define um conjunto de métodos de requisição responsáveis por indicar a ação a ser executada para um dado recurso, também são comumente referenciados como HTTP Verbs (Verbos HTTP). Cada um deles implementa uma semântica diferente.

Vejamos os principais verbos:

GET

O método GET solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados (um JSON por exemplo, veremos esse exemplo em breve).

HEAD

O método HEAD solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta.

POST

O método POST é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.

PUT

O método PUT substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.

Em outras palavras ele faz um update (atualiza um dado existente) de uma série de informações.

Por exemplo:

```
{
  name: ‘Maria’,
  idade: 21,
  estadoCivil: ‘solteira’
}
```

PATCH

Um método de auxílio do PUT. Também tem como objetivo atualizar valores de uma referência, porém o corpo não terá o objeto completo. Sendo assim a atualização dos valores são parciais:

Por exemplo:

```
{
  nome: ‘Maria’
}
```

DELETE

O método DELETE remove um recurso específico.

Respostas HTTP (response)

Os códigos de status das respostas HTTP indicam se uma requisição HTTP foi corretamente concluída. As respostas são agrupadas em cinco classes:

1. Respostas de informação (100-199),
2. Respostas de sucesso (200-299),
3. Redirecionamentos (300-399)
4. Erros do cliente (400-499)
5. Erros do servidor (500-599).

Respostas informativas

100 - Continue: Essa resposta provisória indica que tudo ocorreu bem até agora e que o cliente deve continuar com a requisição ou ignorar se já concluiu o que gostaria.

101 Switching Protocol: Esse código é enviado em resposta a um cabeçalho de solicitação Upgrade pelo cliente, e indica o protocolo a que o servidor está alternando.

102 Processing: Este código indica que o servidor recebeu e está processando a requisição, mas nenhuma resposta está disponível ainda.

103 Early Hints: Este código tem principalmente o objetivo de ser utilizado com o cabeçalho Link, indicando que o agente deve iniciar a pré-carregar recursos enquanto o servidor prepara uma resposta.

Respostas de sucesso (mais comuns)

- GET: O recurso foi buscado e transmitido no corpo da mensagem.
- HEAD: Os cabeçalhos da entidade estão no corpo da mensagem.
- PUT ou POST: O recurso descrevendo o resultado da ação é transmitido no corpo da mensagem.
- TRACE: O corpo da mensagem contém a mensagem de requisição recebida pelo servidor.

200 OK: Estas requisição foi bem sucedida. O significado do sucesso varia de acordo com o método HTTP:

201 Created: A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é uma típica resposta enviada após uma requisição POST/PUT.

Mensagens de redirecionamento

Vou listar apenas uma:

300 Multiple Choice: A requisição tem mais de uma resposta possível. User-agent ou o user deve escolher uma delas. Não há maneira padrão para escolher uma das respostas.

Respostas de erro do cliente

400 Bad Request: Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida.

401 Unauthorized: Embora o padrão HTTP especifique "unauthorized", semanticamente, essa resposta significa "unauthenticated". Ou seja, o cliente deve se autenticar para obter a resposta solicitada.
(Veremos um exemplo no Postman)

404 Not Found: O servidor não pode encontrar o recurso solicitado. Este código de resposta talvez seja o mais famoso devido à frequência com que acontece na web.

Respostas de erro do servidor

500 Internal Server Error

O servidor encontrou uma situação com a qual não sabe lidar. **(Veremos um exemplo no Postman)**

XMLHttpRequest

XMLHttpRequest é um objeto que fornece funcionalidade ao cliente para transferir dados entre um cliente e um servidor. Ele fornece uma maneira fácil de recuperar dados de um URL sem ter que fazer uma atualização de página inteira. Isso permite que uma página da Web atualize apenas uma parte do conteúdo sem interromper o que o usuário esteja fazendo. XMLHttpRequest é usado constantemente na programação de AJAX.

XMLHttpRequest foi originalmente projetado pela Microsoft e adotado pela Mozilla, Apple e Google. Apesar do nome, XMLHttpRequest pode ser usado para recuperar qualquer tipo de dados, e não apenas XML, suportando também, protocolos diferentes de HTTP.

XMLHttpRequest

Vamos ver um exemplo no VsCode:

Fetch

A API Fetch fornece uma interface JavaScript para acessar e manipular partes do pipeline HTTP, tais como os pedidos e respostas. Ela também fornece o método global fetch() que fornece uma maneira fácil e lógica para buscar recursos de forma assíncrona através da rede.

Este tipo de funcionalidade era obtida anteriormente utilizando XMLHttpRequest. Fetch fornece uma alternativa melhor que pode ser facilmente utilizada por outras tecnologias. Fetch também provê um lugar lógico único para definir outros conceitos relacionados ao protocolo HTTP.

Note que a especificação fetch() difere de jQuery.ajax(), principalmente, de três formas:

Fetch

- A Promise retornada do fetch() não rejeitará o status do erro HTTP, mesmo que a resposta seja um HTTP 404 ou 500. Em vez disso, ela irá resolver normalmente (com o status ok definido como falso), e só irá rejeitar se houver falha na rede ou se algo impedir a requisição de ser completada.
- fetch() não receberá cookies cross-site; você não pode estabelecer uma conexão cross-site usando fetch. Cabeçalhos Set-Cookie de outros sites são ignorados silenciosamente.
- fetch() não enviará cookies, a não ser que seja definida a opção credentials do parâmetro init.

Exercício

Encontre na web uma API com verbo GET e tipo de resposta JSON.

Utilize o que viu até agora sobre XMLHttpRequest para consumir essa API e listar alguns dados (pode ser uns 10) no console

Exercício 2

Agora implemente o que desenvolveu no exercício 1 utilizando o objeto Fetch

LocalStorage/SessionStorage

Com o objeto localSotrage e sessionStorage, os aplicativos da web podem armazenar dados localmente no navegador do usuário.

Antes do HTML5, os dados do aplicativo tinham que ser armazenados em cookies, incluídos em todas as solicitações do servidor. O armazenamento na Web é mais seguro e grandes quantidades de dados podem ser armazenadas localmente, sem afetar o desempenho do site.

Ao contrário dos cookies, o limite de armazenamento é muito maior (pelo menos 5 MB) e as informações nunca são transferidas para o servidor.

O armazenamento da Web é por origem (por domínio e protocolo). Todas as páginas, de uma origem, podem armazenar e acessar os mesmos dados.

Diferença entre LocalStorage e SessionStorage

sessionStorage armazena as informações por sessão, ou seja, quando a aba do browser for fechada, as informações salvas em sessionStorage serão apagadas.

Em localStorage os dados permanecerão até que sejam removidos.

Métodos localStorage

localStorage.setItem(‘nomeDaVariavel’, ‘valorDaVarialve’) Obs: os valores sempre devem ser salvos em formato string!

localStorage.getItem(‘nomeDaVariavel’)

localStorage.removeItem(‘nomeDaVariavel’)

Exercício 3

Utilize o exercício anterior para armazenar os dados da API em localStorage. Na próxima vez que a tela for carregada, os dados deverão ser acessados localmente.

Lembrando que se não houver dados locais, sua função deve buscar diretamente na API =)

Boa sorte!

Fetch - Desmistificado!

A resposta retornada pelo fetch trata-se de um objeto Response. Este objeto tem diversos métodos, dentre eles, o método text(), o qual podemos usar para recuperar a informação retornada pela requisição, no formato texto. Este método nos retornará uma Promise que, quando resolvida, nos entregará o conteúdo da requisição.

O objeto Response também tem o método json, que também retorna uma promise, para facilitar caso esteja acessando uma API JSON (no caso do exercício que fizemos). Vamos ver novamente o exercício:

Mas o que é uma Promise?

As Promises definem uma ação que vai ser executada no futuro, ou seja, ela pode ser resolvida (com sucesso) ou rejeitada (com erro).

Há uma diferença entre lançar um erro e rejeitar uma promise. Lançar no erro, vai parar a execução do seu código, é o equivalente a darmos um return em uma função. Porém rejeitar uma Promise fará com que o código continue sendo executado posteriormente.

Promise

```
const promessa = new Promise((resolve, reject) => {
 if(3<2) {
   resolve('msg')
 else{
   reject(e)
}

// Executando uma promise
promessa.then((parametros) => /* sucesso */)
 .catch((erro) => /* erro */)
```

Promise

Como podemos ver, toda a Promise retorna um método then e outro catch , utilizamos o then para tratar quando queremos resolver a Promise, e o catch quando queremos tratar os erros de uma Promise rejeitada. Tanto then quanto catch retornam outra Promise e é isso que permite que façamos o encadeamento de then.then.then .

Para criarmos uma Promise é muito simples, basta inicializar um new Promise que recebe uma função como parâmetro, esta função tem a assinatura (resolve, reject) => {} , então podemos realizar nossas tarefas assíncronas no corpo desta função, quando queremos retornar o resultado final fazemos resolve(resultado) e quando queremos retornar um erro fazemos reject(erro) .

Postman

Até agora só vimos o verbo GET, vamos agora ver um exemplo real do método PUT:

JSON

Vimos até agora várias citações ao JSON, mas o que ele realmente é?

JSON é um formato de troca de informações/dados entre sistemas. Mas JSON significa JavaScript Object Notation, ou seja, só posso usar com JavaScript correto? Não!

O JSON além de ser um formato leve para troca de dados é também muito simples de ler. Mas quando dizemos que algo é simples, é interessante compará-lo com algo mais complexo para entendermos sua simplicidade. Neste caso podemos comparar o JSON com o formato XML.

XML

```
<note>
    <to>Classe</to>
    <from>Gabriel</from>
    <heading>Lembrar</heading>
    <body>
    <laksjdfhlaskdjhf>conteudo</laksjdfhlaskdjhf>
        Não esqueçam de estudar no fds!
    </body>
</note>
```

JSON

```
{
   “id”: 1,
   “nome”: “João”,
   “filhos”: [{“nome”: “filho1”, “idade”: “333999”}, {“nome”: “filho1”, “idade”: “333999”}]
}
```

Vantagens:

1. Leitura mais simples
2. Analisador mais fácil
3. Suporta objetos
4. Velocidade maior na execução e transporte de dados
5. Arquivo com tamanho reduzido
6. Quem utiliza? Google, Facebook, Yahoo!, Twitter...

Exercício

Crie um JSON “pessoa” que contenha seu nome, idade e sexo como atributo. Crie também o atributo caracteristicas que liste suas características físicas, por exemplo “olhos”: “verdes”, cabelo: “loiro”

Salve esse JSON no localStorage.

Na próxima vez que sua tela for carregada, exiba no console as informações dessa pessoa de forma amigável.